IEEE MICRO



A SIMPLE STACK INSTRUCTION SET

JUNE 1984

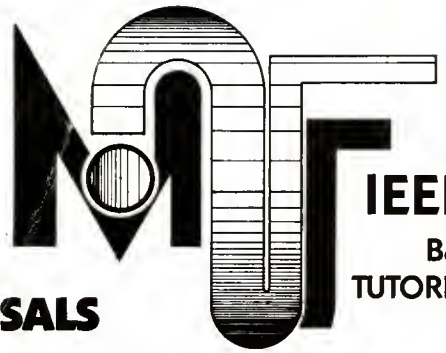
Also in this issue:
- Software driver for the Am9511
- System executive for real-time microcomputer programs
- Proposed standard — IEEE P959 I/O expansion bus

IEEE COMPUTER SOCIETY

THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

1884 1984 A CENTURY OF ELECTRICAL PROGRESS

CALL FOR PAPER PROPOSALS



1985 IEEE MICROPROCESSOR FORUM

Bally's Park Place Casino Hotel — Atlantic City, N.J.
TUTORIALS: March 31-April 1, 1985 FORUM: April 2-4, 1985
EXHIBITS: April 1-4, 1985

Papers are being solicited for the 1985 Microprocessor Forum which is sponsored by the IEEE Computer Society and the IEEE Philadelphia Section. The program will bring together the designers of microprocessor and related chip families with the designers of computer aided engineering (CAE) work stations. The meeting will provide a forum for these designers to share their ideas and experiences regarding the impact of CAE on microprocessor design and conversely the impact of microprocessors on current and future CAE equipment and systems. The program will encompass topics such as systems architecture, hardware design, software design, testability, economic impact trends and extending current technology.

Paper proposals related to but not limited to the following areas are invited:

SILICON BASED ARCHITECTURE
- Architecture
- Memory and memory management
- Input/output processors
- Testability
- Design methodologies
- Simulation
- Design and test algorithms
- Microprogramming

ENGINEERING WORK STATIONS
- System design/architecture for work stations
- Resource sharing — including LANS and distributed networks
- User interfaces for work stations
- Data bases
- Graphics for engineering work stations
- General CAD/CAT/CAI
- Limitations and new requirements for CAE
- Performance/economic impact

EXTENDING TECHNOLOGY THROUGH ENGINEERING WORK STATIONS
- Expert systems/Artifical Intelligence
- Robotics
- Image processing/voice processing/computer vision
- Computer based video
- Wafer integration/system on a chip
- New technologies such as GaAs
- Social implications

DEADLINES:

September 1, 1984 — Receipt of all paper proposals
October 15, 1984 — Authors notified of acceptance
December 15, 1984 — Camera-ready copies of accepted papers is due

INFORMATION FOR AUTHORS:

Submit 3 copies of the paper proposal which should include: (a) paper title; (b) names, affiliations, complete mailing addresses and telephone numbers of all the authors; (c) a summary of about 200 words describing the scope of the paper; (d) a statement of commitment that one of the authors will present the paper at the forum in case of acceptance. A paper lacking any of the above information will not be considered by the Program Committee. The acceptance of papers will be based on technical excellence. The authors must obtain any approvals required to permit presentation of papers.

SUBMIT PAPER PROPOSALS TO:

Walt Luciw
Sperry Corporation
Computer Systems, MSE8-134
P.O. Box 500
Blue Bell, PA 19424
215-542-2760

COMMITTEE

Chairman
Ned Kornfield
Widener University
Chester, PA 19013

Vice-Chairman
Ralph Preiss
IBM Corporation
Poughkeepsie, NY 12602

Treasurer
John Bauer
Naval Air Engineering
Lakehurst, NJ 08733

Secretary
Rich Sweeney
Widener University
Chester, PA 19013

Program Co-Chairmen
Barry Fell
Interspec
Conshohocken, PA 19428

Walt Luciw
Sperry Computer Systems
Blue Bell, PA 19424

Tutorials Chairman
Ken Anderson
Siemens Corporation
Princeton, NJ 08540

Publications
Gerry Gordon
Philadelphia Electric Company
Philadelphia, PA 19101

Publicity
Favio Oliveto
RCA Corporation
Moorestown, NJ 08057

Liaison
Bill Middleton
St. Davids, PA 19087

Arrangements
Bobbie Kornfield
Widener University
Chester, PA 19013

IEEE COMPUTER SOCIETY

1884 1984 A CENTURY OF ELECTRICAL PROGRESS THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

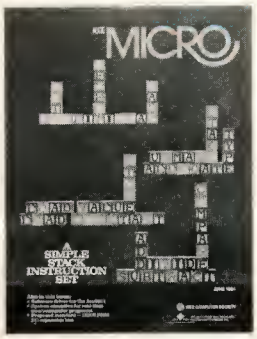
IEEE PHILADELPHIA SECTION

IEEE MICRO

June 1984

Volume 4 Number 3 (ISSN 0272-1732)

ARTICLES



Cover: A Simple Stack Instruction Set—prompted by "A Reduced High-Level-Language Instruction Set," by Peter U. Schulthess (see page 55).

Design: Jay Simpson.

7 An Efficient Software Driver for Am9511 Arithmetic Processor Implementation
Borivoje Furht and *Peter Lee*
This driver enables easy implementation of the Am9511 as a coprocessor in Intel 8085-based systems.

20 A System Executive for Real-Time Microcomputer Programs
Walter S. Heath
Don't want to use an off-the-shelf system executive but lack the resources to develop one from scratch? This description will help you build your own custom package.

33 P959 I/O Expansion Bus—Proposed Standard
I/O Expansion Bus Subcommittee
This proposed standard allows small increments of I/O to be added or modified in an easy, cost-effective way. It is offered here for public comment before submission to the IEEE Standards Board.

55 A Reduced High-Level-Language Instruction Set
Peter U. Schulthess
The Object Pascal Architecture provides 22 simple stack instructions which enable straightforward compilation of Pascal-like languages.

DEPARTMENTS

3 Letters to the Editor
4 Microview: *Factors of production*
68 Micronews
70 Microlaw: *Chip copyright protection*
72 Microreview: *The Sharp PC-5000*
74 Microstandards
75 New Products
81 Product Summary
84 Access
88 Advertiser/Product Index
91 Professional Calendar

Coming in August: Microcomputer Standards

Reader Service Cards, p. 89.

IEEE COMPUTER SOCIETY

EXECUTIVE COMMITTEE

President: Martha Sloan
Department of Electrical Engineering
Michigan Technological University
Houghton, MI 49931
(906) 487-2845

Vice Presidents
Area Activities: Russell E. Theisen
Technical Activities (1st VP): Robert G. Stewart
Conferences and Tutorials (2nd VP): Charles R. Vick
Educational Activities: J. Thomas Cain
Membership and Information: Rolland B. Arndt
Publications: John D. Musa

Treasurer: Helen M. Wood
Secretary: Paul L. Borrill
Junior Past President: Oscar N. Garcia
IEEE Division Directors: Oscar N. Garcia, Merlin G. Smith

GOVERNING BOARD

Term Ending 1984

Marshall D. Abrams
Kenneth R. Anderson
Rolland B. Arndt
James H. Aylor
J. Thomas Cain
Dennis W. Fife
Harvey A. Freeman
Michael C. Mulder
Roy L. Russo
Cyril J. Tunis

Term Ending 1985

Paul L. Borrill
Clyde R. Camp
Herbert Hecht
Glen G. Langdon, Jr.
John F. Meyer
John D. Musa
Harriett B. Rigas
Susan L. Rosenbaum
William R. Shields
Herbert Weber

PUBLICATIONS BOARD

Dharma P. Agrawal
James H. Aylor
Vic Basili
Bruce P. Berra
Bill D. Carroll
Tse-yun Feng
Dennis W. Fife
Paul L. Hazan
Herbert Hecht
Ronald G. Hoelzeman
Glen G. Langdon, Jr.
Theo Pavlidis
C. V. Ramamoorthy
T.R.N. Rao
Peter R. Rony
Susan L. Rosenbaum
Roy L. Russo
Norman Schneidewind
Bruce D. Shriver
Michael J. Wozny
Stephen S. Yau
John D. Musa, Vice President for Publications

SENIOR STAFF

Executive Director: T. Michael Elliott
IEEE Computer Society
1109 Spring Street, Suite 300
Silver Spring, MD 20910
(301) 589-8142
(Will furnish complete roster of committees)

Editor and Publisher: True Seaborn
Director, Conferences and Tutorials: Harry Hayman
Director, Computer Society Press: Chip G. Stockton

Next board meeting:
September 21, Arlington, Virginia

1884 1984 THE INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS, INC.

A CENTURY OF ELECTRICAL PROGRESS

President: Richard J. Gowen
President-Elect: Charles A. Eldon
Executive Vice President: Henry L. Bachman
Executive Director: Eric Herz

IEEE Micro

Editor-in-Chief: Peter R. Rony, VPI & SU*

Associate Editor-in-Chief:
J. Thomas Cain, University of Pittsburgh

Editorial Board:
James H. Aylor, University of Virginia
Patrick Buffet, Digital Equipment Corporation
John S. Burkitt, University of New South Wales
George S. Carson, GSC Associates
James J. Farrell III, Motorola, Inc.
David L. Hannum, AT&T
Kenji Kani, Nippon Electric Company
Henriecus Koeman,
John Fluke Manufacturing Company
L. Robert Morris,
Carleton University and DSPS Inc., Ottawa
Victor P. Nelson, Auburn University
Jean-Daniel Nicoud, Swiss Federal Institute of Technology
Deene Ogden, Texas Instruments
Richard H. Stern, Stern & Roberts
Robert G. Stewart, Stewart Research Enterprises

Magazine Advisory Committee:
Dennis W. Fife (chairman), Dennis R. Allison, Kenneth R. Anderson, Ronald G. Hoelzeman, Stephen F. Lundstrom, Peter R. Rony (ed.-in-chief, *IEEE Micro*), Roy L. Russo (ed.-in-chief, *IEEE Design & Test*), Paul Schneck, Bruce D. Shriver (ed.-in-chief, *IEEE Software*), Michael J. Wozny (ed.-in-chief, *IEEE CG&A*)

Editor and Publisher: True Seaborn
Managing Editor: Joe Schallan

Calendar Editor: Marie English
Contributing Editors: Ware Myers, Charles Lisser
Art Director: Jay Simpson
Production: Kas Winters, Linda Schallan, Roger Winters
Typesetting: Peggy Zeller
Advertising Coordinator: Donald Farnsworth

*Submit six copies of all articles and special issue proposals to Peter R. Rony, Dept. of Chemical Engineering, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061; (703) 961-6370.

Circulation: *IEEE Micro* (ISSN 0272-1732) is published bimonthly by the IEEE Computer Society, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720; (714) 821-8380. Annual subscription: $12.00 in addition to IEEE Computer Society or any other IEEE society member dues. Nonmember prices: available on request. Single-copy prices: members $7.50; nonmembers $15.00. This journal is also available in microfiche form. IEEE Headquarters address: 345 East 47th Street, New York, NY 10017.

Undelivered copies: Send to 10662 Los Vaqueros Circle, Los Alamitos, CA 90720.

Postmaster: Send address changes to *IEEE Micro*, IEEE Service Center, 445 Hoes Lane, Piscataway, NJ 08854. Second class postage is paid at New York, NY, and at additional mailing offices.

Copyright and reprint permissions: Abstracting is permitted with credit to the source. Libraries are permitted to photocopy beyond the limits of US Copyright Law for private use of patrons: those post-1977 articles that carry a code at the bottom of the first page, provided the per-copy fee indicated in the code is paid through the Copyright Clearance Center, 29 Congress St., Salem, MA 01970. Instructors are permitted to photocopy isolated articles for noncommercial classroom use without fee. For other copying, reprint, or republication permission, write to Editor, *IEEE Micro*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720. All rights reserved. Copyright © 1984 by the Institute of Electrical and Electronics Engineers, Inc.

Editorial: Unless otherwise stated, bylined articles, as well as products and services offered in New Products, the Product Summary, Microreview, Micronews, and Access, reflect the author's or firm's opinion; inclusion in this publication does not necessarily constitute endorsement by the IEEE or the IEEE Computer Society.

BPA

LETTERS TO THE EDITOR

Microprocessors defy classification

The designation of a computer as 8-bit, 16-bit, or 32-bit seems to be more a marketing decision by the manufacturer than anything else. Every definition I have heard results in an absurd designation for some computer. For example, that offered by James Isaak in the December 1983 issue of *IEEE Micro* (Letters, page 3) would designate the IBM System/370 as a 12-bit computer.

There is a consensus in the personal computer marketplace that the IBM PC is a 16-bit computer. Those writing about such systems have a difficult time explaining how the 8086 differs from the 8088.

Hubert Kirrmann's rather complex classification (December 1983, Letters, page 4) does seem to capture the essence of current microprocessors. The designation is usually applied to the architecture of a processor rather than its implementation, however. In his definition, both numbers are implementation-dependent, though the first is usually determined by architecture. The introduction of on-chip caches—already announced by several manufacturers—will soon render even this definition inadequate.

James R. Goodman
Computer Sciences Dept.
University of Wisconsin—Madison
1210 West Dayton Street
Madison, WI 53706

Author's reply:

The most important parameter of a processor seems to me to be its address size. Therefore, I proposed to classify a microprocessor by both its address and its data size and thus designated a 68000 a 24+16 processor and an 8086 a 20/16 processor (the slash means that address and data are multiplexed, while the plus sign means that they are not). Goodman is right to observe that a cache makes this definition inadequate to describe perfor-

cont't on page 6

Most Program Editors Are Shockingly Primitive.



Use Pmate™ once, and you'll never go back to an ordinary text editor again. Pmate is more than a powerful programmer's text processor. It's an interpretive language especially designed for customizing text processing and editing.

Just like other powerful editors, Pmate* features full-screen single-key editing, automatic disk buffering, ten auxiliary buffers, horizontal and vertical scrolling, plus a "garbage stack" buffer for retrieval of deleted strings. But, that's just for openers.

What really separates Pmate from the rest is macro magic. A built-in macro language with over 120 commands and single-keystroke "Instant Commands" to handle multiple command sequences. So powerful, you can "customize" keyboard and command structure to match your exact needs.

Get automatic comments on code. Delete comments. Check syntax. Translate code from one language to another. Set up menus. Help screens. You name it.

And, Pmate has its own set of variables, if-then statements, iterative loops, numeric calculations, a hex to decimal and decimal to hex mode, binary conversion, and a trace mode. You can even build your own application program right inside your text processor.

So, why work with primitive tools any longer than you have to? Pmate by Phoenix. $225. Call 1-800-344-7200. Or Write.

Phoenix

Phoenix Software Associates Ltd.
1420 Providence Highway Suite 260
Norwood, MA 02062
In Massachusetts (617) 769-7020

*Pmate is designed for microcomputers using the Intel 8086 family of processors, and running MS-DOS.™ A custom version is available for the IBM PC, TI Professional, Wang Professional, DEC Rainbow, and Z80 running under CP/M.™

Pmate is a trademark of Phoenix Software Associates Ltd.
MS-DOS is a trademark of Microsoft Corporation. CP/M is a trademark of Digital Research, Inc.

Reader Service Number 20

microVIEW



Norman Y. Mineta

Factors of production

by Congressman Norman Mineta

Government can provide an environment that is conducive to the success of business, and government can also erect obstacles which make success impossible. For example, the operations of business require public utilities, public educational systems, publicly funded research, and publicly regulated channels of transportation and communication. But skewed governmental tax policies, misguided laws, and barriers to free competition can doom productive business. Willy nilly, high-technology business enterprises are inextricably linked to the policies of their government.

As a member of Congress who represents part of California's Silicon Valley, I have seen that the semiconductor and electronics industries touch upon many of the most intriguing and contentious political questions of our day: military spending, social spending, monetary policy, trade policy, industrial policy, educational policy.

Consider some of the most basic public policy questions now pending both here in the United States and in Europe and Japan. Should your government spend large portions of public revenues on military projects? Some may gain business from defense contracts, but others may suffer because America has a shortage of engineers for nonmilitary projects. Moreover, others are hurt by the higher interest rates and inflated prices which result from high levels of military spending and bloated deficits.

Should the government maintain, raise, or lower taxes? None of us likes giving up money. But business depends upon the goods and infrastructure provided with public funds. It depends upon roads and airports built with tax revenues. It depends upon technological advances provided by tax-subsidized research. It depends upon people whose training has been assisted by tax revenues.

And, of course, we must ask ourselves questions about trade policies. The problem is to protect ourselves against unfair trade practices without depriving ourselves of the benefits of free trade. Many employees are unable to move to another field or another section of the country; how should we deal with the fact that this week's laid-off employees may not be the same person as next week's new-hire? The fact is that people are not infinitely flexible and redeployable factors of production. Perhaps there are cases when our government should protect a declining firm or industry to lessen human dislocation.

Some people prefer to avoid the arguments which these questions provoke. But the issues will not fade away. So there is plenty of controversy. But I believe that useful public policy depends upon the successful discovery of consensus. Despite the questions I have raised, I believe that there is a great deal of room for agreement in the policy issues affecting high-tech industries and in particular the electronics industry. Moreover, these realms of agreement have led to several current useful legislative proposals in the United States Congress.

Research and development: antitrust. First and foremost we can agree that research and development provide the lifeblood on innovative, growing industries. Firms in the electronics industry are buffeted in the tides of rapid technological change, and their survival often depends upon their ability to rise with those tides. Here in the United States, our government has sometimes proven insensitive to their situation. As a result, entrepreneurs may face some senseless prohibitions of antiturust law, some unreasonable taxes, and burdens because of insufficient funding for education and research.

Efforts to adjust U.S. antitrust laws, in order to permit joint R&D vantures, offer us further room for agreement and productive action. Many U.S. companies have already joined together in R&D ventures, but the antitrust laws have not yet been changed to ensure that these companies will be free from burdensome and unnecessary antitrust challenges from other private parties or from the government. We Americans have witnessed the successes of foreign competitors who have been able to cooperate in basic R&D ventures, and these successes provide further strong arguments for promoting such associations in the U.S. By allowing our domestic companies to work together on R&D projects, our companies will be able to avoid unnecessary duplication of effort and inefficient use of resources.

Just how the antitrust laws should be changed is still being debated. Some of the proposals currently being considered include measures which would provide full or limited immunity from antitrust prosecutions or a safety-net provision limiting antitrust penalties to actual damages or single-damages instead of the present triple-damage awards. Some proposals call for "safe harbors" if the venture can satisfy standards assuring probable lack of significant anticompetitive effects. The idea of the "safe harbor" is to permit businesses to undertake certain joint R&D projects with the knowledge that because the projects meet predetermined minimum standards assuring that anticompetitive results are unlikely, investments can safely be made in the projects without concern that antitrust problems will subsequently undermine the project. I think all of the current proposals deserve study, and in principle they warrant support. Again, I believe that after hearing and considering the views of those concerned we can coalesce in agreement about the need for some reforms of U.S. antitrust laws.

R&D: tax. These are fields calling for new congressional initiatives. For example, to spur investment in R&D, many of us in Congress supported legislation providing tax credits for investment in research and development. In 1981, such legislation was enacted. Fortunately, the credits provided by that legislation have been highly successful in promoting new product development. But the credits

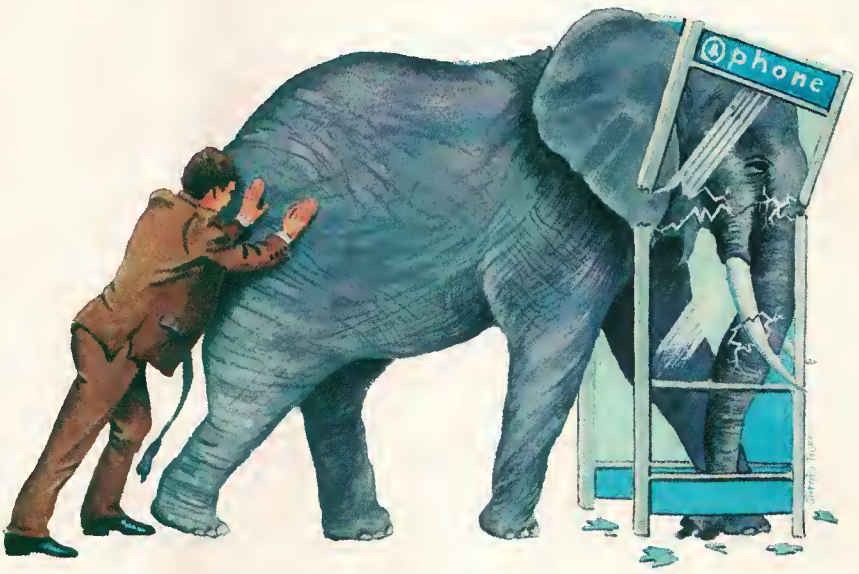
were provided only on a temporary basis, so it is important that support be displayed for proposed legislation for the development of new products which is now before Congress—legislation to extend the R&D tax credit and make it a permanent part of our tax code. I support that legislation. Moreover, it is important that the legislation that emerges gives fair treatment to all important forms of R&D, such as computer software as well as hardware.

Even the sticky issue of taxes can, I believe, invite productive agreement. Policy makers in Washington have begun to appreciate the fact that while President Reagan's Economic Recovery Act of 1981 provided a windfall for traditional and "smokestack" industries, it did little for high-technology industries. In 1981, too few understood that high-technology companies depend upon relatively short-lived equipment to survive in a constantly changing industry. As a result of that oversight, the 1981 Recovery Act provided tax breaks for traditional, capital-intensive, "smokestack" industries—tax breaks which were nearly irrelevant for the high-technology electronics industries. It is now time to cooperate in an effort to get such industries some reasonable depreciation provisions in the U.S. tax code—provisions which will benefit high-growth electronics companies.

Chip protection. Representing Silicon Valley, I have also become acutely conscious of the fact that the special nature of R&D for silicon chips requires further legislative reforms. Representative Don Edwards and I have introduced the Semiconductor Computer Chip Protection Act (H.R. 1028) in the U.S. House of Representatives to address this fact. Our bill provides much needed copyright protection for the layouts and mask patterns of computer chips. Legal acknowledgement of the property rights of the owner of a chip layout embodied in mask designs will ensure that the time and money spent by a company to develop a new chip will be a worthwhile investment. Innovating firms often spend years, thousands of engineering manhours, and millions of dollars in order to develop a new large-scale integrated semiconductor circuit design. Yet a competing firm can photograph a chip and thus duplicate the masks used to make it, in only a few months and for a cost as low as $50,000. Once again, I believe that we can agree—we can agree on the basic principle that reasonable copyright protection is both fair and productive. And that principle can help move us toward the legislative reform embodied in the Semiconductor Chip Protection Act.

Education. Of course, neither incentives for R&D nor tax credits for rapidly

Squeezing A Large Program Into A Small Memory Space?



It's time you got Plink86,™ the overlay linkage editor that's bringing modular programming to Intel 8086-based micros.

With Plink86,* you can write a program as large and complex as you want and not worry about whether it will fit within available memory constraints. You can divide your program into any number of tree-structured overlay areas. 4095 by 32 deep. Work on modules individually. Then link them into executable files. All without making changes to your source program modules.

Use the same module in different programs. Experiment with changes to the overlay structure of an existing program. Use one overlay to access code and data in other overlays.

Plink86 is a two-pass linkage editor that links modules created by the Microsoft assembler or any of Microsoft's 8086 compilers. Plink86 also works with other popular languages, like Lattice C, C86, or mbp/COBOL. And supports symbolic debugging with Phoenix' Pfix86 Plus.™

Plink86 includes its own object module library manager – Plib86™ – that lets you build libraries from scratch. Add or delete modules from existing libraries...Merge libraries... Or produce cross-reference listings.

Why squeeze any more than you have to? Plink86 by Phoenix. $395. Call (1) 800-344-7200. Or, write.

Phoenix

Phoenix Software Associates Ltd.
1420 Providence Highway Suite 260
Norwood, MA 02062
In Massachusetts (617) 769-7020

*Plink86 will run under PC DOS, MS-DOS™ or CP/M™-86.

Plink86, Pfix86 Plus and Plib86 are trademarks of Phoenix Software Associates Ltd.
MS-DOS is a trademark of Microsoft Corporation. CP/M is a trademark of Digital Research, Inc.

Reader Service Number 21

depreciating capital can take full, beneficial effect if our electronics companies do not have the trained personnel necessary to utilize opportunities in high-tech industries. There can be no doubt that the international technological revolution has caught the United States understaffed and surprisingly undereducated. You may have heard about the American Electronics Association report which concluded that America's educational system needs to triple its output of electrical and computer science engineers in the next five years if it is going to meet the needs of the electronics industry alone. Indeed, it is staggering to consider the small number of people on whom the United States depends for its technological expertise. For example, out of 200 million Americans, only 1000 are trained to design large-scale integrated circuits.

In recognition of such deficiencies, the United States House of Representatives has passed the Math and Science Education Act of 1983. This bill authorizes funds to improve and increase math and science education at all levels in America, to update facilities in our university laboratories, and to promote stronger ties between research in our universities and innovations in our private industries. That bill is currently being considered in the Senate, and I hope that we can also cooperate in an effort to get the Math and Science Education Act passed in Congress and signed into law.

Yet we must do more than offer seed money to improve our academic environment. The U.S. government will have to offer more than that to help update our educational systems and provide the crucial braintrust necessary for the success of your industries. I am now supporting measures in the House of Representatives to provide tax credits to companies donating computers and other scientific equipment to our nation's schools. That is probably only one of many bills which could get private companies to help improve the technological education provided by our schools. For example, I feel that the federal government could and should cooperate in some way with private firms in the effort to provide a sufficient supply of high-calibre educators. Concerned citizens in Silicon Valley have begun to worry about the "Brain Drain" developing as firms hire away vitally needed professors. I think we can all agree that Congress, private industry, and the educational community should work together to help preserve our nation's precious supply of teachers, so that we will have people who will train the next generation of scientists and technologists—and not, in effect, eat our own seed corn.

Trade. There is another point on which we can all agree—that is that without international trade, the electronics industries would suffer immensely. And if all the measures I have discussed are to bear their full and fair dividends, we must also address the state of world trade itself. The current administration has imposed severe and unnecessary restrictions on the export of many technological products, thousands of products which our military adversaries neither want nor need and which are readily available from other sources. As a result, business people have to wade through a sea of bureaucratic red tape. Again, I think we can agree on appropriate legislative redress. We should adjust the legislation which allows all of these unnecessary and detrimental restrictions, the Export Administration Act. In Congress, we have been working to amend that legislation so that only the trade restrictions which truly promote our national security will remain in place. Here again, we should remind those who hesitate that our national security depends upon vigorous trade as well as military might.

Involvement. All of these initiatives address specific concerns of the electronics industries. But I think it is important also to try to grapple with a wider array of public policy issues. As the high-technology lobby matures, there may be a great temptation to think only about devising a menu of tax credits or public incentives which affect your balance sheets. But it must be recognized that all decisions about high-technology issues in the public sector will have to be made within the context of larger decisions about our future. Therefore, just as government provides essential factors of production for industry, concerned citizens must provide the essential factors of production for government. We must all involve ourselves in the broadest issues of our day, such as foreign affairs, the provision of health care, and the maintenance of our environment. All of these deserve your attention. Government needs your energy and your expertise.

Norman Y. Mineta (D) represents California's 13th Congressional District, located at the southern tip of San Francisco Bay. The district includes portions of San Jose and the Cities of Campbell, Los Gatos, and Santa Clara, as well as unincorporated parts of the County of Santa Clara.

Born in 1931, Mineta was one of 110,000 Americans of Japanese ancestry who were evacuated from the West Coast and placed in relocation camps during WWII. Attending schools in San Jose, he later graduated from UC Berkeley in 1953 with a BS in business, then served on active duty in the Army as an intelligence officer in Japan and Korea.

Commencing his political career in 1962, Mineta served in a variety of local offices including membership on the San Jose City Council and mayor of San Jose. He has held a seat in Congress since 1974, serving on several major committees ranging from the Public Works and Transportation Committee to the Select Committee on Intelligence—posts he continues to hold today. Currently he is also a member of the House Committee on Science and Technology, where he serves on the Subcommittee on Space Science and Applications and the Subcommittee on Science, Research, and Technology.

Letters

cont'd from p. 3

mance. If the 8086 (20/16) and the 8088 (20/8) both had an on-chip cache with a hit ratio of 95 percent, the performance difference between them would matter only five percent of the time. The hardware designer must be aware of the difference all of the time, however. Goodman is also right to point out that my classification does not necessarily reflect the architecture as the programmer sees it. Even the external address size can be controversial—the Microvax's address size of 28 bits is larger than the physically available 22-bit address on the Q-bus. Single-board computers and signal processors are difficult to describe. Yet, the address and data that come out of the chip (or out of the board) are good indicators of the architecture and of the implementation, respectively.

It is difficult to describe a microprocessor with only a few symbols. To integrate more parameters, one should also indicate the size of the on-chip memory and the clock speed—one should call the Z80000 a "32/32 C256 K5" processor, for instance, since it has a multiplexed 32-bit address and data path, a 256-byte cache, and a 5-MHz bus clock (which is more meaningful than the CPU clock). The 8051 would be a "17/8 R4K + M128 K12," since it has two address spaces, 4K of ROM plus 128 bytes of RAM, and a 12-MHz bus clock. Such a method of description would be rather cumbersome, of course.

Hubert Kirrmann
Brown, Boveri & Cie
Research Center
CH-5405 Baden, Switzerland

This driver enables easy implementation of the Am9511 as a coprocessor in Intel 8085-based systems.

An Efficient Software Driver for Am9511 Arithmetic Processor Implementation

Borivoje Furht, University of Miami

Peter Lee, IBM, Corp.

An arithmetic processor unit interfaced to a host microprocessor system as a coprocessor significantly improves system performance, performing both a variety of fixed and floating point arithmetic operations and trigonometric and mathematical operations. An efficient software driver has been developed that enables easy implementation of the arithmetic processor unit (an Am9511 or Intel 8231A) in a concurrent operation with a host microprocessor (e.g., Intel 8085).

Basic hardware configuration

The Am9511 arithmetic processing unit can be interfaced to an Intel 8085/8086 host microprocessor just as any programmable I/O unit can. All transfers between the microprocessor and the arithmetic processing unit are performed using an 8-bit directional data bus. Transfers to and from the APU are handled by using a conventional programmed I/O technique. To provide I/O mapped I/O an 8205 decoder is used to get the chip select to the APU with IO/$\overline{M}$ signal equal 1. The signals $\overline{RD}$ and $\overline{WR}$ are tied directly to the APU. Since the upper address lines (A_8-A_{15}) are non-multiplexed, they are used for selection of the APU. This technique relies on the fact that the I/O port number is copied onto A_8-A_{15} as well as A_0-A_7 during an input or output instruction. The basic hardware interface scheme is shown in Figure 1.

The signal $\overline{PAUSE}$ from the APU is tied to the READY line of the 8085 to avoid data loss. This line is high in its initial state and is pulled low by the APU under the following conditions:

- A previously initiated operation of the APU is in progress and a command entry is attempted.
- A previously initiated operation of the APU is in progress and stack access has been attempted.
- The APU is not busy and data removal or data entry has been requested.

Am9511 operation

The Am9511 APU operation is based on commands supplied by a host microprocessor. Each command represents a single 8-bit datum, shown in Figure 2.

- Bits 0-4 select the operation of the APU.
- Bits 5-6 select the data format for the operation. If bit 5 = 1, then a fixed-point datum is specified. If bit 5 = 0, floating-point format is specified. Bit 6 selects the precision of the data. If bit 6 = 1 single precision (16 bits) is selected, while bit 6 = 0 selects double precision (32 bits).
- Bit 7 indicates whether a service request is to be issued after the command is executed. If bit 7 = 1, a service request will be generated; if bit 7 = 0, a service request will not be generated.

0272-1732/84/0600-0007$01.00 © 1984 IEEE

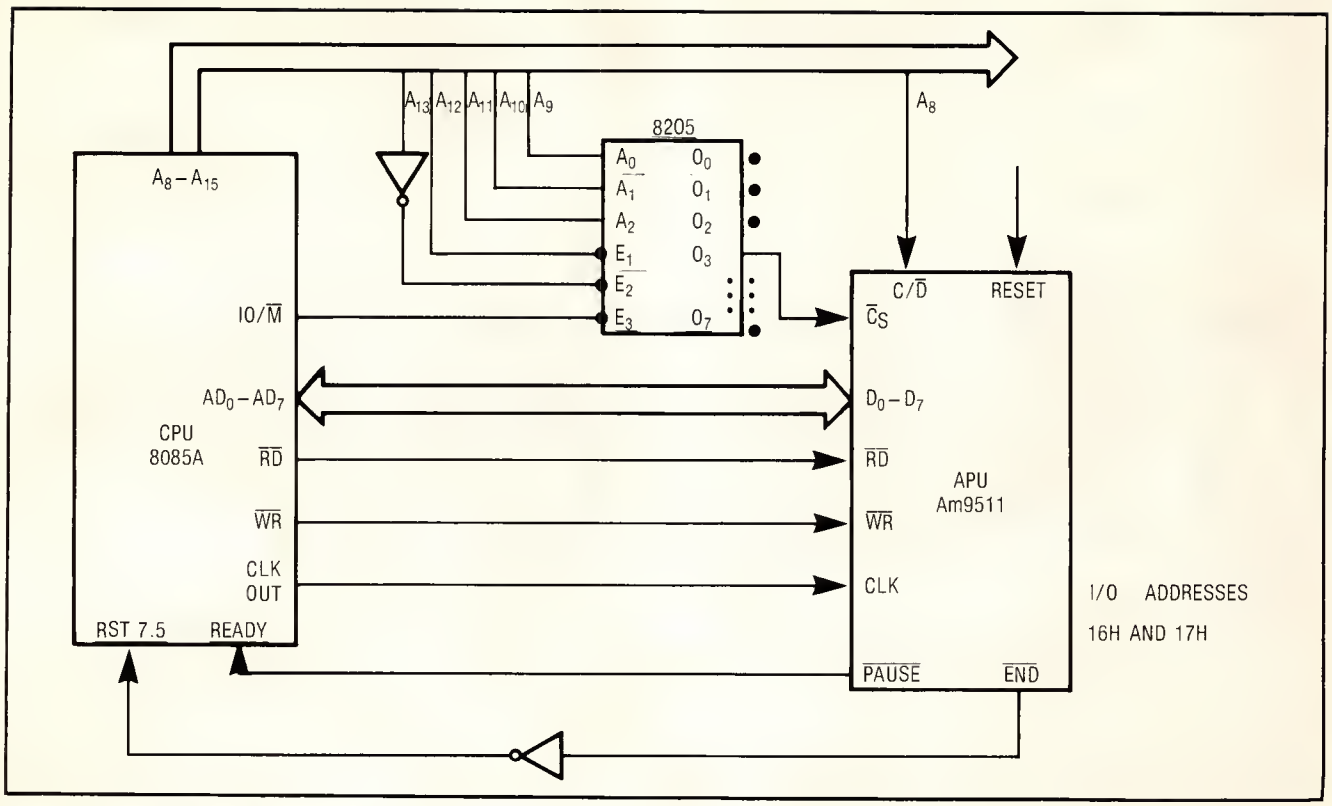


Figure 1. Basic hardware interface.

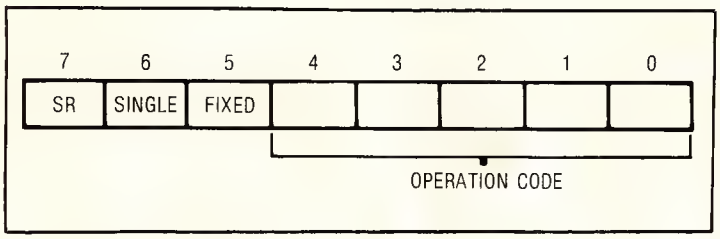


Figure 2. Command format.

A summary of the APU commands is given in Table 1. (In Table 1, TOS means top of stack and NOS means next of stack. In the hex-code column, SVREQ is a 0.

The Am9511 APU is a stack-oriented processor, so the host processor has access to an eight-level, 16-bit-wide data stack. Single-precision fixed-point operands are 16 bits wide, so eight such values may be stored in the stack (Figure 3a). When the data are double-precision fixed-point or floating-point operands (32 bits), the four values may be stored (Figure 3b). Data are written into the stack byte after byte, in the order shown in Figure 3 (B1, B2, B3, . . .). Data are removed from the stack in reverse order (B8, B7, B6, . . .).

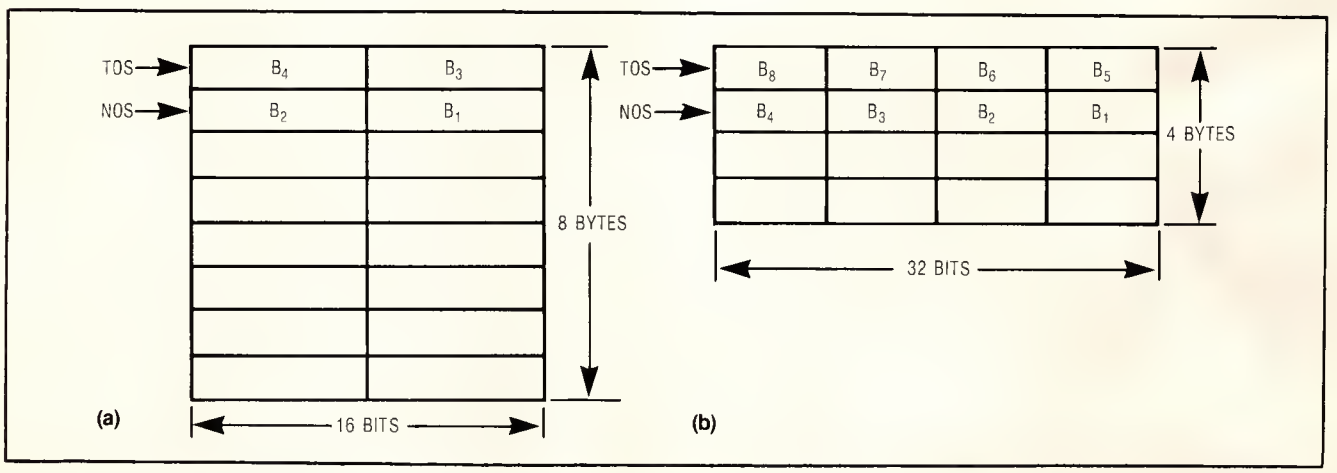


Figure 3. Structure of the APU's stack: (a) single-precision fixed-point operands; (b) double-precision fixed-point or floating point operands.

Execution times

Timing for execution of the Am9511 commands is shown in Table 2. Speeds are given in terms of clock cycles and can be converted to microseconds by multiplying by the clock period used. Maximum clock frequency for the Am9511 is 2 MHz and for the Am9511-1 is 3 MHz. One version of the Intel 8231A is faster (4 MHz).

Interface strategies

Two different interface techniques have been applied: pseudopolling and interrupt-driven interfaces.

Pseudopolling interface technique. In this mode of operation the host processor sends one command after another to the APU using a standard output operation. Since a previously initiated operation will usually be in progress when a new command entry is attempted, the APU will pull the $\overline{\text{PAUSE}}$ line low, causing the host processor to skip into a waiting state. The $\overline{\text{PAUSE}}$ line will remain low until completion of the current command execution. It will then go high, permitting entry of the new command.

The software driver for the pseudopolling mode of operation (SDPPM) is simple and consists of a set of subroutines that send commands to the APU, have access to the APU stack, and remove/enter data from/to the

Table 1.
Command summary.

COMMAND MNEMONIC	COMMAND DESCRIPTION	HEX COMMAND CODE
	FIXED POINT 16 BIT	
SAOO	Add TOS to NOS. Result to NOS. Pop Stack.	6C
SSUB	Subtract TOS from NOS. Result to NOS. Pop Stack.	60
SMUL	Multiply NOS by TOS. Lower half of result to NOS. Pop Stack.	6E
SMUU	Multiply NOS ty TOS. Upper half of result to NOS. Pop Stack.	7C
SDIV	Divide NOS by TOS. Result to NOS. Pop Stack.	6F
	FIXED POINT 32 BIT	
DAOO	Add TOS to NOS. Result to NOS. Pop Stack.	2C
OSUB	Subtract TOS from NOS. Result to NOS. Pop Stack.	2D
DMUL	Multiply NOS by TOS. Lower half of result to NOS. Pop Stack.	2E
OMUU	Multiply NOS by TOS. Upper half of result to NOS. Pop Stack.	3C
OOIV	Oivide NOS by TOS. Result to NOS. Pop Stack.	2F
	FLOATING POINT 32 BIT	
FAOO	Add TOS to NOS. Result to NOS. Pop Stack.	10
FSUB	Subtract TOS from NOS. Result to NOS. Pop Stack.	11
FMUL	Multiply NOS by TOS. Result to NOS. Pop Stack.	12
FOIV	Oivide NOS by TOS. Result to NOS. Pop Stack.	13
	DERIVED FLOATING POINT FUNCTIONS	
SORT	Square Root of TOS. Result in TOS.	01
SIN	Sine of TOS. Result in TOS.	02
COS	Cosine of TOS. Result in TOS.	03
TAN	Tangent of TOS. Result in TOS.	04
ASIN	Inverse Sine of TOS. Result in TOS.	05
ACOS	Inverse Cosine of TOS. Result in TOS.	06
ATAN	Inverse Tangent of TOS. Result in TOS.	07
LOG	Common Logarithm (base 10) of TOS. Result in TOS.	08
LN	Natural Logarithm (base e) of TOS. Result in TOS.	09
EXP	Exponential (e^X) of TOS. Result in TOS.	OA
PWR	NOS raised to the power in TOS. Result in NOS. Pop Stack.	OB
	DATA MANIPULATION DEMANDS	
NOP	No Operation	00
FIXS	Convert TOS from floating point to 16-bit fixed-point format.	1F
FIXO	Convert TOS from floating point to 32-bit fixed-point format.	1E
FLTS	Convert TOS from 16-bit fixed-point to floating-point format.	10
FLTD	Convert TOS from 32-bit fixed-point to floating-point format.	1C
CHSS	Change sign of 16-bit fixed-point operand on TOS.	74
CHSO	Change sign of 32-bit fixed-point operand on TOS.	34
CHSF	Change sign of floating-point operand on TOS.	15
PTOS	Push 16-bit fixed-point operand on TOS to NOS (Copy).	77
PTOO	Push 32-bit fixed-point operand on TOS to NOS (Copy)	37
PTOF	Push floating-point operand on TOS to NOS (Copy).	17
POPS	Pop 16-bit fixed-point operand from TOS. NOS becomes TOS.	78
POPD	Pop 32-bit fixed-point operand from TOS. NOS becomes TOS.	08
POPF	Pop floating-point operand from TOS. NOS becomes TOS.	38
XCHS	Exchange 16-bit fixed-point operands TOS and NOS.	39
XCHO	Exchange 32-bit fixed-point operands TOS and NOS.	39
XCHF	Exchange floating-point operands TOS and NOS.	19
PUPI	Push floating-point constant "π" onto TOS. Previous TOS becomes NOS.	1A

Table 2.
Command execution times.

COMMAND MNEMONIC	CLOCK CYCLES	COMMAND MNEMONIC	CLOCK CYCLES
SADD	17	SQRT	800
SSUB	30	SIN	4464
SMUL	84-94	CDS	4118
SMUU	80-98		
SDIV	84-94	TAN	5754
DADD	21	ASIN	7668
DSUB	38	ACOS	7734
DMUL	194-210	ATAN	6006
DMUU	182-218	LOG	4474-7132
DDIV	208		
FIXS	92-216	LN	4298-6956
FIXD	100-346	EXP	3794-4876
FLTS	98-186	PWR	8290-12032
FLTD	98-378		
FADD	54-368	NDP	4
FSUB	70-370	CHSS	23
FMUL	146-168	CHSD	27
FDIV	154-184	CHSF	18
PTDS	16	PDPF	12
PTOD	20	XCHS	18
PTOF	20	XCHD	26
POPS	10	XCHF	26
PDPD	12	PUPI	16

APU stack. The software driver SDPPM, written in assembly language for the 8085, requires 256 bytes of memory.

A typical command routine for floating point multiplication is shown below:

```
FMUL:  MVI A, 12H   ; Code for multiplication
       OUT 17H      ; Send command to the APU
       RET
```

A typical entry routine ENT16 for entering a 16-bit value into the APU stack has the following form:

```
ENT16: MOV A, M
       OUT 16H      ; Enter low byte
       INX H        ; Increment pointer
       MOV A, M
       OUT 16       ; Enter high byte
       RET
```

Before calling a routine for data entry/removal, the user should load the register pair H&L with the address of data.

The user program is a simple sequence of CALL instructions to the corresponding APU routines.

Example 1. The following example illustrates the implementation of the SDPPM by a user program. Suppose the following computation should be performed as

$$y = \sqrt{x \cdot \sin x}$$

where x is a 16-bit integer.

Here is an efficient way to perform the desired function:

- Load the 16-bit integer value of x in the APU.
- Execute APU instruction FLTS to convert to floating point.
- Execute APU instruction PTOF to duplicate the result of the floating-point conversion.
- Execute APU instruction SIN to replace the value of x at the top of the stack with its sine.
- Execute APU instruction FMUL to compute $x \cdot \sin x$.
- Execute APU instruction SQRT to compute square root of $x \cdot \sin x$.
- Finally, remove the result from the APU stack.

The user program that performs the computation of y is the following sequence of CALL instructions:

```
LXI   H, XDATA  ;Pointer to x
CALL  ENT16     ;Load 16-bit data into the APU
CALL  FLTS      ;Convert to floating point
CALL  PTOF      ;Duplicate x
CALL  SIN       ;Sin(x)
CALL  FMUL      ;x·sin(x)
CALL  SQRT      ;Square root of x·sin(x)
LXI   H, YDATA  ;Pointer to y
CALL  REM32     ;Remove 32-bit result from the
                 APU
```

The execution time analysis of the given example is shown in Figure 4.

As can be seen from Figure 4, the microprocessor is in a waiting state during the operation of the APU, so a concurrent operation of the μP and the APU is not realized.

Interrupt interface technique. In this mode of operation, the signal $\overline{\text{END}}$, generated by the APU, is used to activate an interrupt service routine in the μP.

The host microprocessor, after preparing a set of commands and the corresponding data for the APU, is able to continue the execution of its own program.

In order to enable a concurrent operation of the μP and the APU, the following strategy, consisting of two steps, has been implemented. First, the user commands to the APU, given in a form of macroinstructions, are not executed directly but are temporarily stored in a command buffer in RAM. Also, the corresponding data are not entered directly onto the APU stack, but its addresses-pointers to the data are temporarily stored into a pointer buffer in RAM. The command buffer is 20 bytes long, so the user can write an operation consisting of a maximum of 20 APU commands. The data buffer is 10 bytes long, so five 16-bit pointers to the data can be stored in it.

In the second step, at the end of a macro sequence, the user activates an interrupt service routine by using a CALL instruction. The interrupt service routine coordinates the execution of the commands previously stored in the command buffer. Upon completion of each command from the command buffer, the APU will issue an "end of execution signal" that will again activate the interrupt service routine. Then the ISR will send the next command from the command buffer to the APU.

Software driver for the interrupt mode. In defining software requirements for an APU software driver for the interrupt mode of operation (SDIM), the main objectives were

- to make the use of the Am9511 easy, efficient, and flexible from the user's point of view, and
- to provide a concurrent operation of the host μP and the APU.

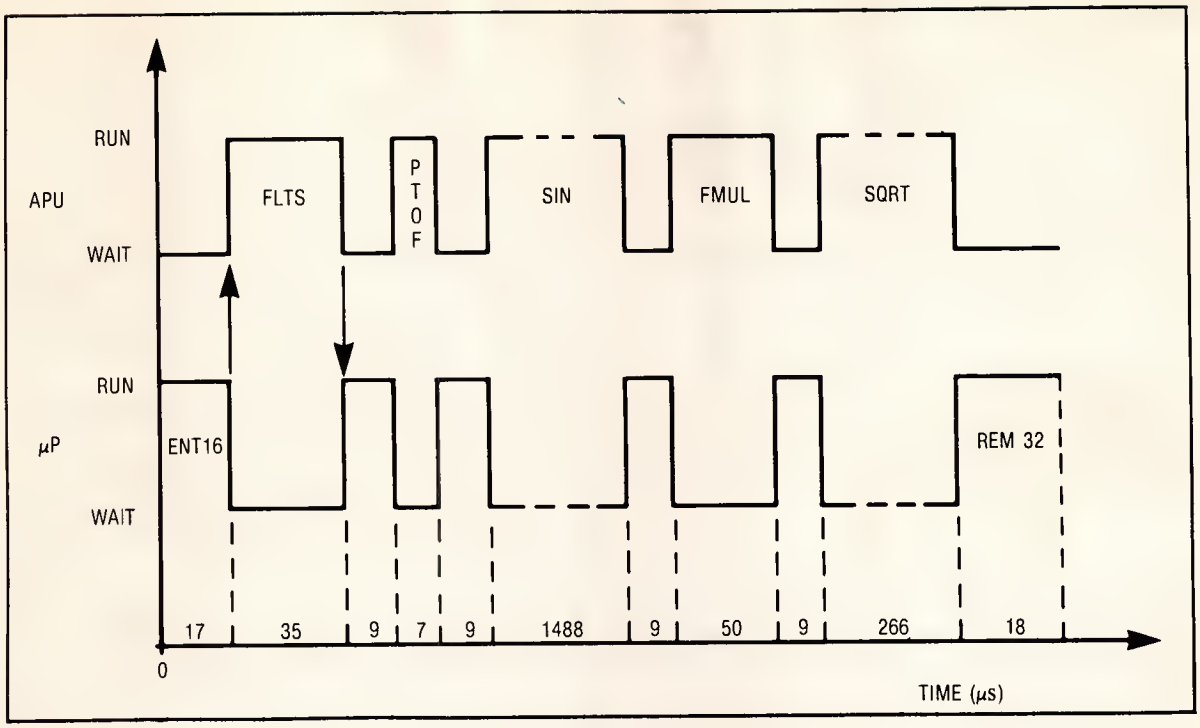


Figure 4. Execution time analysis: pseudopolling technique.

The designed software-driver SDIM consists of a set of macro definitions that perform all necessary data/command preparation and an interrupt service routine that coordinates commands execution.

Data entry/removal macros. There are two data-entry and two data-removal macroinstructions that initialize 16-and 32-bit data transfer. These macroinstructions store the pointers to the data into the pointer buffer and an internal code for the corresponding command (entry or removal) into the command buffer.

For example, macroinstruction ENT16 DATAX stores the pointer to the data, with a symbolic address DATAX, into the pointer buffer, and the command 40H, that corresponds to 16-bit data-entry into the command buffer. The macro definition for ENT16 has the following format:

```
ENT16 MACRO DATA
      LXI   D, DATA  ;Pointer to the data . . .
      MOV   M, E
      INX   H
      MOV   M, D
      INX   H        ; . . . store into the PB
      MVI   A, 40H   ;Code store into . . .
      STAX  B
      INX   B        ; . . . the CB
      ENDM
```

The internal codes for the entry/removal instructions are as follows:

Instruction	Hex Code
ENT16	40
ENT32	41
REM16	50
REM32	51

Register pairs HL and BC are used as the pointers to the pointer buffer and the command buffer, respectively. In order to initialize any sequence of the APU operations, the user has to set these pointers to the top of the pointer buffer and the command buffer by using the macroinstruction INIT at the beginning of an APU sequence. Also, the macro INIT clears the command buffer and the pointer buffer. The macro definition INIT has the following format:

```
INIT    MACRO PBUF,CBUF
        LXI   H,PBUF    ;Clear the pointer buffer
        MVI   D,10
        XRA   A
LOOP1:  MOV   M,A
        INX   H
        INR   D
        JNZ   LOOP1
        LXI   B,CBUF    ;Clear the command
                         buffer
        MVI   D,20
LOOP2   STAX  B
        INX   B
        INR   D
        JNZ   LOOP2
        LXI   H,PBUF    ;Pointer to the top of
                         the pointer buffer
        LXI   B,CBUF    ;Pointer to the top of
                         the command buffer

        ENDM
```

Command entry macros. There are 42 command entry macro definitions that store the corresponding hex code (see Table 1) into the next available location of the command buffer and then increment the pointer to the command buffer. For example, macro definition FMUL for floating-point multiplications has the following format:

```
FMUL MACRO
     MVI    A,92H  ; Code for multiplication
     STAX   B      ; Store code into the
                     command buffer
     INX    B      ; Increment pointer to
                     the command buffer
     ENDM
```

All the other APU commands have the same structure of macro definition.

Interrupt service routine. The interrupt service routine coordinates the execution of the commands previously stored in the command buffer. The ISR also accomplishes data entry onto the APU stack and data-removal from the APU stack, using the pointers to the data previously stored in the pointer buffer. The current pointer to the data is always stored at the top of the pointer buffer, and the current command code is always stored at the top of the command buffer. After completion of a data entry, the current pointer will be removed and the next pointer from the pointer buffer will be shifted to the top of the pointer buffer. Similarly, after completion of a command entry to the APU, the command code from the command buffer will be shifted to the top of the command buffer. This avoids the necessity of saving pointers of the pointer buffer and the command buffer.

The flowchart of the ISR is shown in Figure 5, and the corresponding program is given in the appendix. When the last command from an APU sequence has been completed, the ISR will set an "end of the APU sequence" flag in order to inform the main program of the completion of the whole APU operation. This flag can be at any location in memory. In the designated SDIM the flag is at the top location of the pointer buffer.

Example 2. In order to illustrate the implementation of the SDIM, consider the same expression $y = \sqrt{x \cdot \sin x}$ given in Example 1. The user program is a sequence of macro references given below. At the beginning the user should reserve memory locations for the pointer buffer and command buffer.

```
        ; Memory reservation
PBUF:   DS     10     ; Pointer buffer
CBIF:   DS     20     ; Command buffer
X:      DS     2      ; 16-bit data
Y:      DS     4      ; 32-bit result
        ; The APU sequence y=square root(x·sinx)
START:  INIT          ; Initialization
        ENTI6  X      ; Enter 16-bit data
        FLTS          ; Convert to floating-
                        point
        PTOF          ; Duplicate X
        SIN           ; Sin x
        FMUL          : x·sin x
        SQRT          ; Square root of x·sin
        REM32         ; Remove 32-bit data
        CALL   ISR    ; Call interrupt service
                        routine
                      ; to activate the APU
                        operation
```

After execution of the whole sequence of macroinstructions, the APU is not yet activated; however, the pointer buffer contains the pointers to the data x and y, and the command buffer contains the codes of the used APU commands, as shown in Figure 6.

In order to compare the described interrupt-interface technique and the pseudopolling technique, the execution time analysis for Example 2 is shown in Figure 7.

It is obvious from Figure 7 that the microprocessor is never in a waiting state, so the microprocessor and the APU operate concurrently. After preparing a set of APU commands and the corresponding data in the command buffer and the pointer buffer, the host microprocessor is able to continue the execution of its own program.

In the case when microprocessor program needs a result from a previously activated APU sequence, it can test the "end of the APU sequence" flag set by the ISR at the end of an APU sequence.

APU implementation in a multimicroprocessor system

In recent years the cost of microprocessor components has been reduced, so the concept of applying multiple processors to achieve better system performance, previously avoided, has now become an attractive and feasible design strategy. Since the cost of an APU is still high, however, including more than one APU in an MMP system would be very expensive.

In this section we propose an MMP configuration with multiple processors that share a single APU. The APU software driver described can be used in this MMP system.

A multiple master/dual bus multimicroprocessor architecture has been applied and described by Adams and Rolander.[4] Each processor-master has its own private memory and I/O. Access to the common bus occurs only when a master requires access to the common memory or common I/O. A parallel bus priority technique has been used which enables up to eight masters to run concurrently. The control logic for the common bus access was designed using an Intel 8219 bus-controller.

An Am9511 APU is a common I/O unit shared by all processors. The corresponding APU software drivers are the same for all processors and stored in their local memories. When a master requires an APU operation, it executes the corresponding APU driver from its local memory and activates the APU.

After completing a single operation, the APU sends an interrupt signal to the corresponding master.

The architecture of an MMP system with common APU is shown in Figure 8. Note that there needs to be additional logic to allow each processor select to either the local or common bus. Showing the details of the selection would make the figure complicated, so they are omitted.

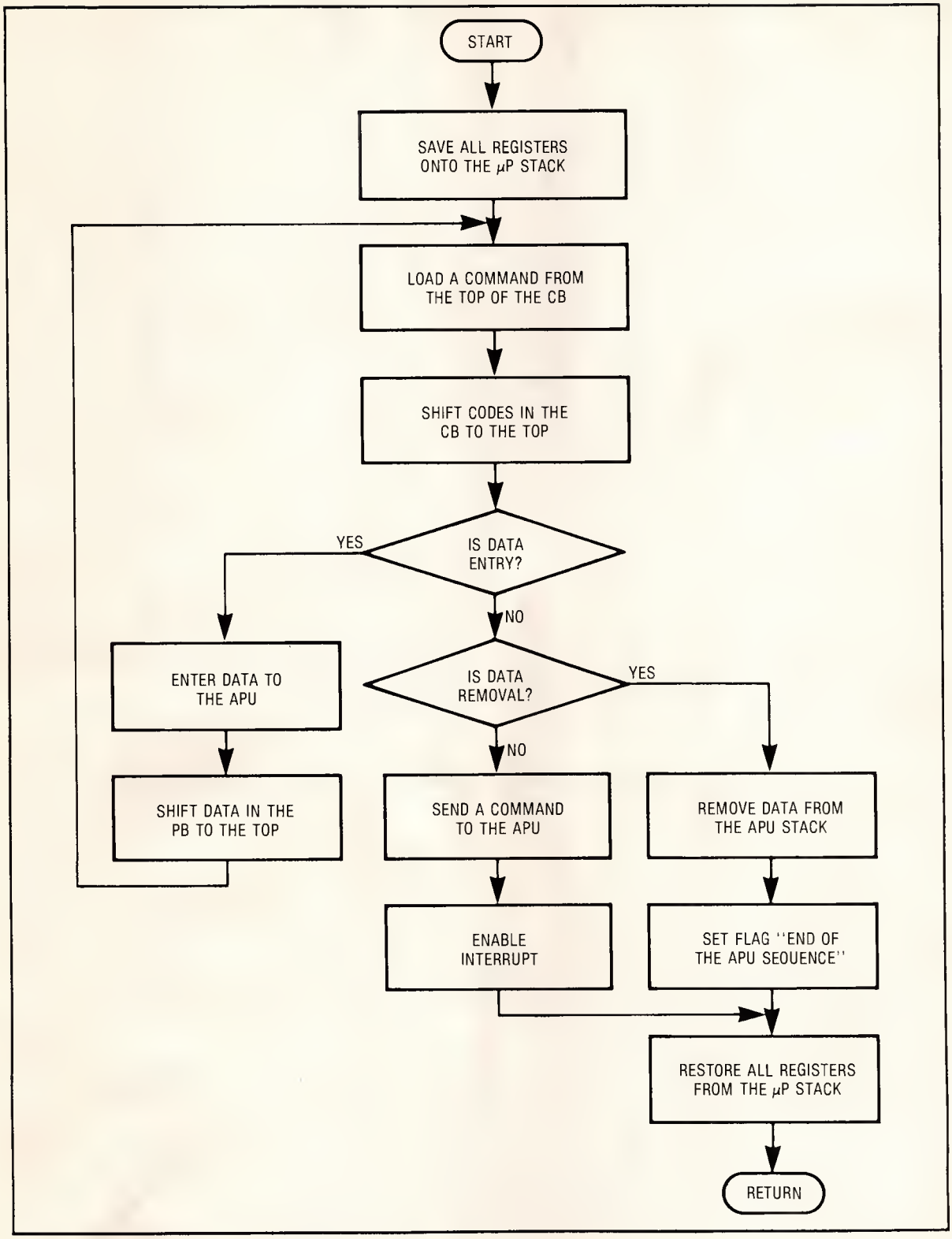


Figure 5. The interrupt service routine.

Two problems have to be solved in the MMP configuration. The first problem is mutual exclusion. It is necessary to assure that a master will have uninterrupted access to the APU. Therefore, a critical section of code has been introduced which, once begun, must complete its execution without interruption. This critical sequence has been controlled by using the hardware/software solution described below. The second problem is the interrupt signal, sent by the APU after completion of its operation, that is interfaced to all the masters (Figure 8). In order to cause the ac-

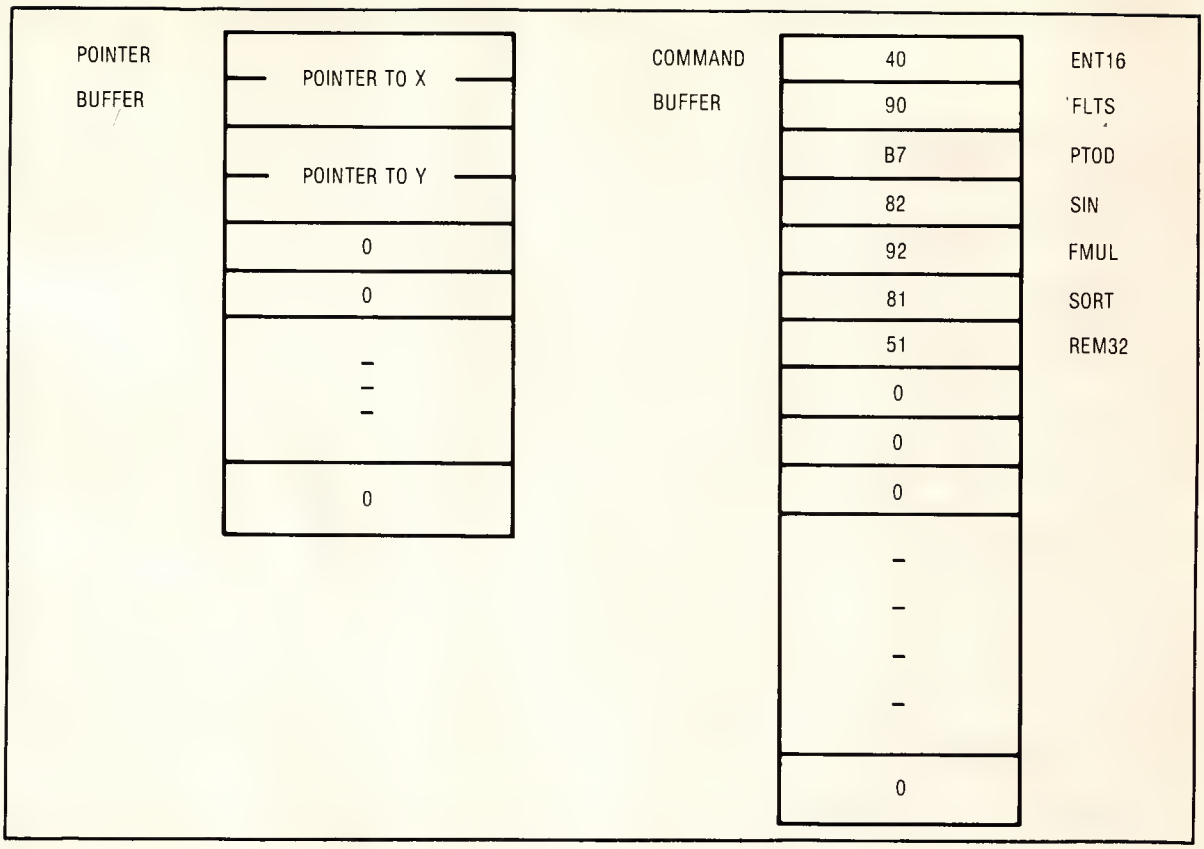


Figure 6. The contents of the pointer buffer and the command buffer in Example 2.

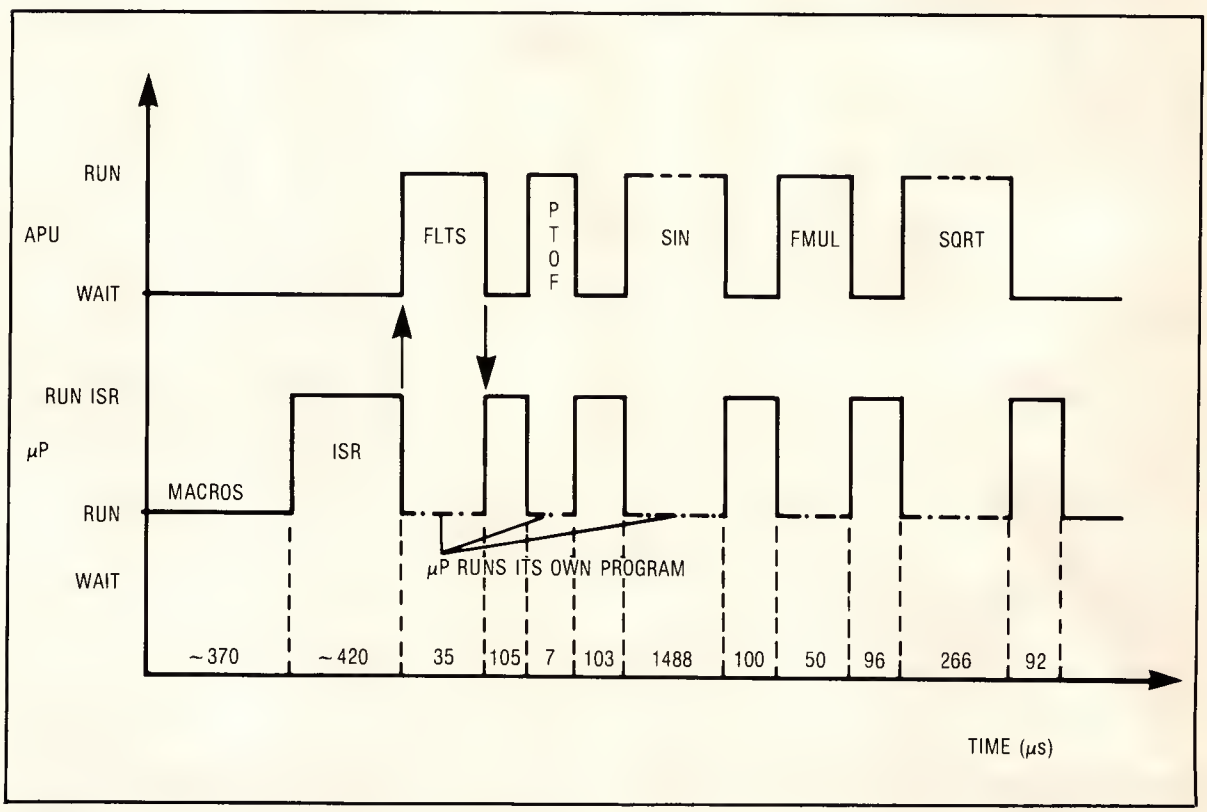


Figure 7. Execution time analysis: interrupt technique.

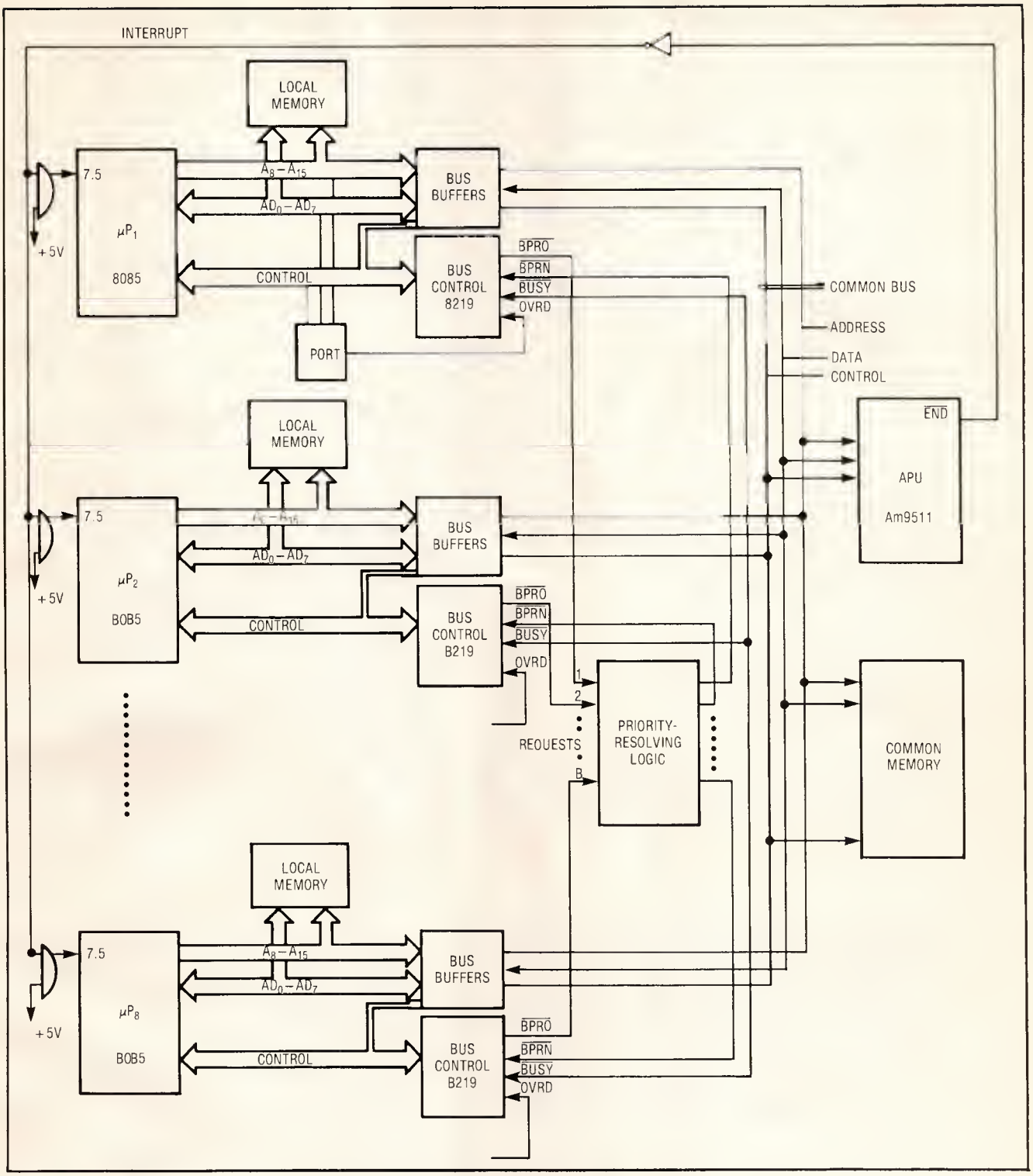


Figure 8. Multimicroprocessor system with a common APU.

tivation of the interrupt routine in the master which had activated the APU, a software interrupt-mask technique should be used.

The software driver SDIM, described in the previous section, can be easily extended in order to be used in the proposed MMP system with a common APU. First, a subroutine TEST (see Figure 9) is introduced into the driver that controls the critical section of code and that solves the mutual exclusion problem. The routine TEST should be called by the user program after the preparation of the commands has been completed and before the execution of the ISR.

Mutual exclusion is controlled by an "APU busy flag" in the common RAM memory set by the processor which

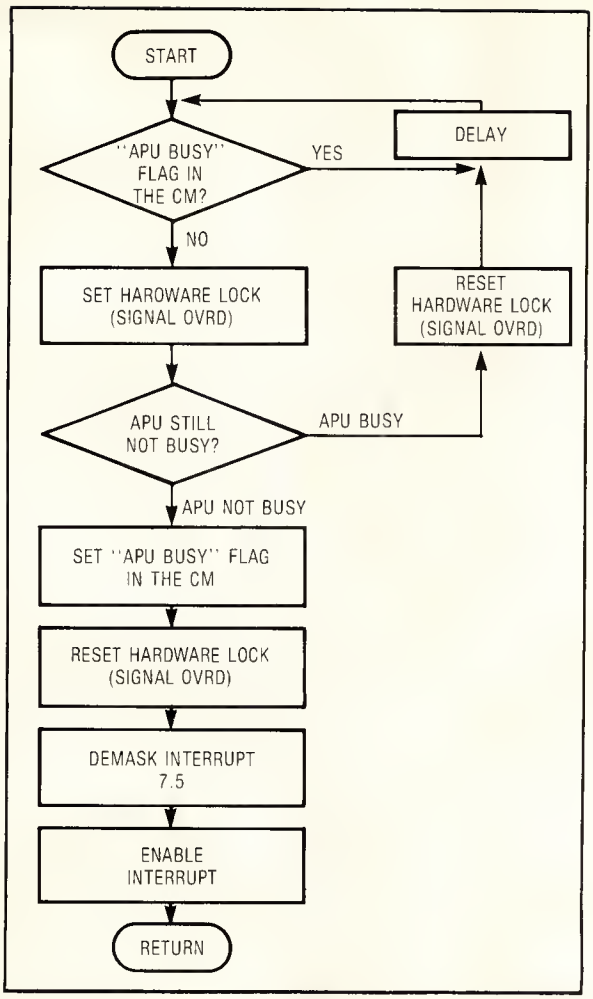


Figure 9. The routine TEST that controls the critical section of code.

uses the APU, and using a hardware signal "Override" generated by the microprocessor through an output port. The override signal temporarily locks the access of the other processors to the common bus.

Finally, the APU interrupt routine ICR should also be modified in order to be applied in the MMP system. The "APU busy" flag should be set at the beginning of the ISR, and the same bit should be reset at the end of an APU sequence. At the same time, the interrupt mask for 7.5 should be set in order to disable the activation of the ISR in the processor.

Example 3. A simple example illustrates the implementation of the described software driver in an MMP system in which two microprocessors share an APU.

Suppose that processor 1 (P1) should perform the following operation:

$$x = a*b$$

while processor 2 (P2) performs the following operation:

$$y = c*d$$

The user programs, stored in the local memories of P1 and P2, are as follows:

PROCESSOR	**1**	**PROCESSOR**	**2**
INIT		INIT	
ENT16	A	ENT16	C
FLTS		FLTS	
PLTD		PLTD	
ENT16	B	ENT16	D
FLTS		FLTS	
FMUL		FADD	
REM32	X	REM32	Y
CALL	TEST	CALL	TEST
CALL	ISR	CALL	ISR

Each microprocessor can be in one of the three possible states:

- running state (RUN), executing its main program;
- ISR-state, executing its interrupt APU service routine;
- waiting state, when the APU is busy executing an APU sequence of another processor.

The APU itself can be in either RUN or WAIT state. The timing diagram, shown in Figure 10, illustrates the operation of two processors that share an APU.

- Suppose that at time t_1 processor 1 starts the execution of an APU sequence, preparing the commands and the data.
- At time t_2, P_1 has completed the execution of its macro sequence, storing all necessary commands and data into its command and pointer buffer, respectively. It executes the routine TEST and, since the APU is not busy, it starts the execution of the APU commands. Because of the use of this mutual exclusion technique, the access to the APU is inhibited for the other processors (in this example, for processor 2).
- At time t_3 processor 2 also starts the execution of its APU sequence, preparing the corresponding commands and data.
- At time t_4, P2 starts the execution of its TEST routine, testing the "APU busy" flag in the common memory. Since the APU is still busy executing the sequence of P1, P2 starts the execution of a waiting loop, shown in Figure 9.
- At time t_5, the APU operation of P1 has been completed, so the ISR of P1 will clear the "APU busy" flag in the CM. P2, which was in a waiting loop testing this flag, will obtain an access to the APU and will start the execution of its APU sequence.

This analysis can be extended to the case when several processors share an APU.

The APU software driver in a high-level language environment

The APU software driver SDPPM can be easily modified and used in a high-level language environment.

Intel's 8085 Floating-Point Arithmetic Library contains basic floating-point subroutines and functions referred to as procedures.[6] The FPAL can be used by assembly language or PL/M programs.

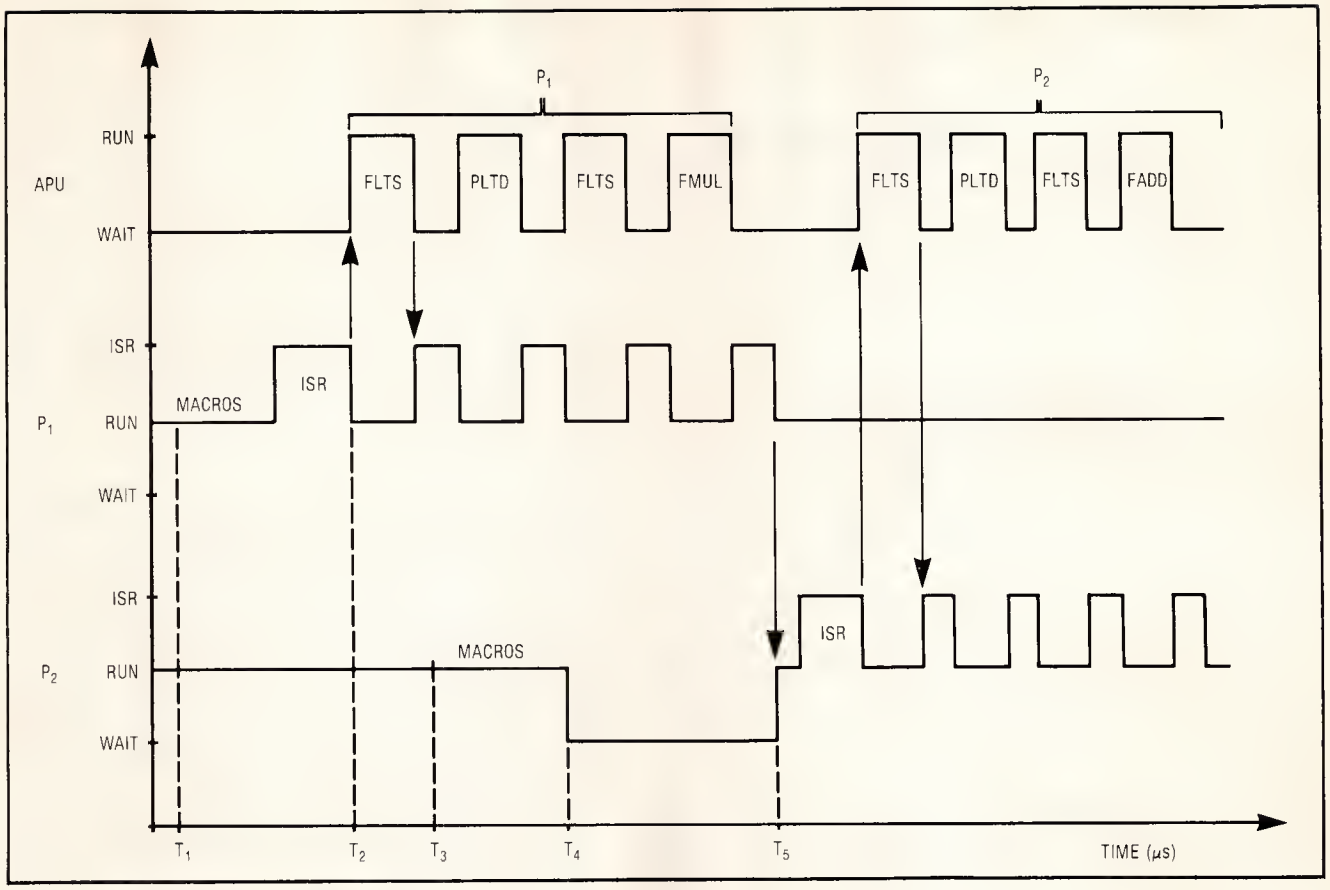


Figure 10. Timing diagram—MMP system.

We have implemented the same strategy in designing an APU software driver to a high-level language, or SDHLL. A set of the procedures has been designed that use the APU to perform the arithmetic operations, instead of doing this by software routines. From the user's point of view, the APU is invisible and the use of the SDHLL routines in a PL/M program is the same as the use of the FPAL routines.

In general, the following steps should be observed to use the SDHLL:

- An area of memory must be reserved for the floating-point record.
- The names of the SDHLL procedures the user plans to use must be declared to be "external" using the EXTERNAL attribute in PL/M-80.
- The SDHLL procedure references must be embedded in the user's source code where appropriate.
- The SDHLL procedure used by user's program should be linked to the user's object file.

Example 4. The following PL/M example computes the function

$$F = A*B+C$$

where A, B, and C represent addresses of the floating-point numbers. F is the address where the result is to be stored, and FPR is the address of the floating-point register.

We must first declare the SDHLL procedures used to be external and reserve the FPR memory as an array. The PL/M program is given below:

```
/* EXAMPLE 4 - F = A * B + C */
/* ---------------------------------------- */
/* DEFINE EXTERNAL PROCEDURES */
FSET: PROCEDURE (FA,0P1,0P2) EXTERNAL;
      DECLARE (FA,0P1,0P2) ADDRESS;
END FSET;
FADD: PROCEDURE (FA,0A) EXTERNAL;
      DECLARE (FA,0A) ADDRESS;
END FADD;
FMUL: PROCEDURE (FA,0A) EXTERNAL;
      DECLARE (FA,0A) ADDRESS;
END FMUL;
FLOAD: PROCEDURE (FA,0A) EXTERNAL;
       DECLARE (FA,0A) ADDRESS;
END FLOAD;
FSTOR: PROCEDURE (FA,0A) EXTERNAL;
       DECLARE (FA,0A) ADDRESS;
END FSTOR;
/* DECLARE BYTE ARRAYS */
DECLARE FPR(18) BYTE,
            A(4)    BYTE,
            B(4)    BYTE,
            C(4)    BYTE,
            F(4)    BYTE;
```

Table 3.
Execution times: SDHLL vs. FPAL.

PROCEDURE	AVERAGE EXECUTION TIME (MICROSECONDS) FPAL	SDHLL
FADD	700	30
FSUB	800	35
FMUL	1500	50
FDIV	3700	57

```
/* COMPUTATION OF F = A*B + C*/
/* FSET SHOULD BE CALLED FIRST */
        CALL FSET(.FPR,0,0);        /*Initialization of FPR*/
        CALL FLOAD (.FPR,.A);       /*Load data A*/
        CALL FMUL (.FPR,.B);        /*Multiply A and B*/
        CALL FADD (.FPR,.C);        /*Compute A*B*+C*/
        CALL STOR (.FPR,.F);        /*Store result in F*/
```

The SDPPM driver can be modified in order to be consistent to the existing FPAL library package. A detailed description of the SDHLL is given by Furht and Milutinovic.[7]

The existing PL/M programs, which use the FPAL routines, can implement the SDHLL routines without any change. In general, the only change occurs during the linking process. When the user has completed program development, he should link the SDHLL, instead of the FPAL, to his object modules.

It is obvious that execution speeds of SDHLL procedures are much faster than those of the corresponding FPAL procedures. Table 3 gives a brief comparison of the execution speeds of some commonly used procedures. The microprocessor clock frequency is supposed to be 6MHz and the APU clock frequency is 3MHz.

These software drivers enable easy and efficient APU implementation both in single microprocessor systems and in a multimicroprocessor system with a common APU. Two different interface techniques have been used: pseudopolling and interrupt techniques. The corresponding APU drivers have been developed—the SDPPM and the SDIM, respectively. The SDPPM can be modified for use in a high-level language environment. Using the APU driver, the user can focus attention on the problem by simply calling a set of macroinstructions or "CALL procedures" to activate the APU functions.

The APU software driver could be included in a compiler for a language such as PL/M, Pascal, or Ada.■

Appendix: The Interrupt service routine of the SDIM

```
; ***********************************************
; INTERRUPT SERVICE ROUTINE FOR SDIM
  -RST 7.5
; ***********************************************
; First activation: CALL instruction from the main
;                   program at the end of an APU
;                   sequence.
; Next activation: Interrupt signal END whenever
;                  a single APU operation has been
;                  completed.
; Function:        Send a new command/data to
;                  the APU using the top of
;                  command/pointer buffers.
; APU addresses: 16H - Enter/read byte into/from
;                  the APU.
;                17H - Enter command/ read
;                  status

ISR:    PUSH    PSW         ; Save registers
        PUSH    B
        PUSH    D
        PUSH    H
ISRO:   LDA     CBUF        ; Load next
                            command from
                            the CB
        MOV     B,A         ; Save command
                            in B
        LXI     H,CBUF      ; Pointer to the
                            top of the CB
        LXI     D,CBUF+1    ; Pointer to next
                            address
ISR1:   LDAX    D           ; Shift codes in
                            the CB . . .
        MOV     M,A
        ORA     A
        JZ      ISR2
        INX     H
        INX     D
        JMP     ISR1        ; . . . to the top
                            until 0 code is
                            found
ISR2:   MOV     A,B         ; Restore current
                            command code
        ANI     0F0H        ; Mask most
                            significant 4
                            bits
        CPI     40H
        JZ      ISR3        ; Test if data
                            entry command
        CPI     50H
        JZ      ISR6        ; Test if data
                            removal
                            command
; Command entry to the APU
        MOV     A,B         ; Restore
                            command code
        OUT     17H         ; Send to the
                            APU
        EI                  ; Enable
                            interrupt
        JMP     ISR9        ; Jump to the
                            end
; Data entry to the APU
ISR3:   LHLD    PBUF        ; Address of the
                            first byte
        MOV     A,M         ; Send 1. byte
        OUT     16H
        INX     H           ; Address of the
                            second byte
        MOV     A,M         ; Send 2. byte
```

```
        OUT     16H
        MOV     A,B         ; Restore
                            command code
        CPI     41H
        JNZ     ISR5        ; If 16-bit data
        INX     H           ; Address of the
                            third byte
        MOV     A,M         ; Send 3. byte
        OUT     16H
        INX     H           ; Address of the
                            third byte
        MOV     A,M         ; Send 3. byte
        OUT     16H
ISR5:   LXI     B,CBUF      ; Shift pointers
                            in the PB . . .
        LXI     D,CBUF+2    ; . . . to the top
                            until 0 pointer
                            is found
ISR5A:  LDAX    D
        STAX    B
        MOV     L,A         ; LSB of the
                            address being
                            moved
        INX     D
        INX     B
        LDAX    D
        STAX    B
        INX     D
        INX     B
        ORA     L           ; MSB address is
                            currently in A
        JNZ     ISR5A
        JMP     ISR0        ; Jump to "Load
                            next
                            command"
```

Data removal from the APU stack in reverse order

```
ISR6:   LHLD    PBUF        ; Address of the
                            result in HL
        MOV     A,B         ; Restore
                            command word
        CPI     51H
        JZ      ISR7        ; If 32-bit result
        INX     H           ; End of the
                            16-bit data
                            buffer
        IN      16H
        MOV     M,A         ; Store 1. byte
        DCX     H
        IN      16H
        MOV     M,A         ; Store 2. byte
        JMP     ISR8
ISR7:   INX     H
        INX     H
        INX     H           ; End of 32-bit
                            data buffer
        IN      16H
        MOV     M,A         ; Store 1. byte
        DCX     H
        IN      16H
        MOV     M,A         ; Store 2. byte
        DCX     H
        IN      16H
        MOV     M,A         ; Store 3. byte
        DCX     H
        IN      16H
        MOV     M,A         ; Store 4. byte
ISR8:   MVI     A,1         : Set flag "End
                            of APU
                            operation"
        STA     PBUF
ISR9:   POP     H           ; Restore
                            registers
        POP     D
        POP     B
        POP     PSW
        RET
```

References

1. "Am9511 Arithmetic Processor Unit," data sheet, Advanced Micro Devices, Inc., Sunnyvale, CA, Mar. 1978.
2. B. K. Gupta, "Arithmetic Processor Chips Enhance Microprocessor Performance," *Computer Design,* Vol. 19, No. 7, July 1980, pp. 85-94.
3. E. B. Croson, F. H. Carlin, and J. A. Howard, "Integrated Arithmetic Processing Unit Enhances Processor Execution Times," *Computer Design,* Vol. 20, No. 4, Apr. 1981, pp. 163-171.
4. G. Adams and T. Rolander, "Design Motivations for Multiple Processor Microcomputer Systems," *Computer Design,* Vol. 17, No. 3, Mar. 1978.
5. R. O. Parker and J. H. Kroeger, "Algorithm Details for the Am9511 Arithmetic Processing Unit," Pub. No. AM-PUB072, Advanced Micro Devices, Inc., Sunnyvale, CA, 1978.
6. *8085/8085 Floating-Point Arithmetic Library User's Manual,* Pub. No. 9800452B, Intel Corp., Santa Clara, CA, 1977.
7. B. Furht and V. Milutinovic, "The Am9511 Arithmetic Processor Implementation in a High-Level Language Environment," in progress.



Borivoje Furht is an associate professor in the Electrical and Computer Engineering Department of the University of Miami, Coral Gables. From 1970 to 1981, he was with the Computer System Division of the Institute "Boris Kidric" —Vinca, Yugoslavia, where he was a project leader in designing specialized microprocessor-based systems and their application in telecommunication and control. Since 1982, he has been an assistant professor of electrical and computer engineering at the University of Miami. He has done research in computer architecture, multi-microprocessor systems, supercomputers, and software engineering. He is currently designing a super microcomputer used in image and signal processing.

Furht is the author of more than 40 technical papers and articles, about 30 technical reports, and two books. He is a member of IEEE and its Computer Society and a member of the Editorial Board of the *International Journal of Mini and Microcomputers.* He received the BSc, MSc, and PhD degrees in electrical and computer engineering from the University of Belgrade, Yugoslavia, in 1970, 1973, and 1978, respectively.

Furht's address is Department of Electrical and Computer Engineering, University of Miami, P.O. Box 248294, Coral Gables, Florida 33124.



Peter Lee is a member of the technical staff at IBM in Boca Raton, Florida, involved in design and development of advanced computers. Earlier he was on the technical staff at Gould, Inc, SEL. He received his BSEE in 1980 and MSEE in 1982 from the University of Miami.

Lee's address is IBM, Computer Systems Division, 2000 51st Street, Boca Raton, Florida 33431.

Don't want to use an off-the-shelf system executive but lack the resources to develop one from scratch? This description will help you build your own custom package.

A System Executive for Real-Time Microcomputer Programs

Walter S. Heath

The availability of inexpensive microprocessors and support chips has proven to be a mixed blessing to the designers of electronic systems that must respond immediately to external events. On the one hand, systems that were previously implemented using customized, discrete logic may now be designed using a general-purpose microprocessor, thus reducing initial circuit design costs and improving the ability of the system to respond to subsequent design changes. Indeed, the flexibility of these devices has made possible the design of systems with logical complexities far exceeding those of earlier designs using discrete elements.

On the other hand each new technological advance brings its own problems, and the microprocessor is no exception. Designers of microprocessor-based systems have, no doubt, experienced the high costs and delays inherent in producing the software. The cost of producing customized hardware logic has been replaced by the cost of producing custom software logic. This has not been a zero-sum game, however; the cost per logical operation has been reduced substantially. Rather, it is the sheer number and complexity of the logical operations performed in the software that has increased costs. In addition, the cost of maintaining the software as changes are made also increases as the initial structure of the program becomes lost in the changes. Clearly, if reasonably priced systems of increasing logical complexity are to be produced in the future, attention must continually be focused on methods for producing the software more efficiently.

A traditional method for increasing software productivity has been to use a high-level language, such as Fortran or Cobol. This approach has worked well for scientific and business programming applications where either the speed of execution could be sacrificed or large, fast computers could be used. Indeed, the relatively slow Basic interpreter may be credited with much of the success in applying microcomputers to various data processing tasks. However, for applications in which speed of execution has been important, designers have had to remain with assembly language. This situation is changing for some applications.

As computer language designers develop new, structured languages and efficient compilers, and as microprocessors are designed to support these new languages and run at faster clock-rates, it becomes more reasonable to consider using a high-level language in time-critical applications. Indeed, the benefits may justify the use of multiple microprocessors running in parallel to achieve system timing requirements. Reduced initial software development and subsequent maintenance costs can be considerable.

Another way to improve software productivity and run-time efficiency is to provide an "environment" within which application programs may be executed in an efficient and timely manner. In real-time systems this environment is usually called a "system executive" and the application programs are referred to as "tasks" (or processes). The system executive is generally responsible for

0272-1732/84/0600-0020$01.00 © 1984 IEEE

making sure application tasks are executed at the proper times and for providing a formal means for tasks to communicate with each other.

Tasks must also be able to communicate with the outside world. An input/output device may be serviced by polling it periodically and testing its status or by allowing it to cause an interrupt. Polling, however, wastes computer processing time and introduces delays. For time-critical applications this approach is therefore generally inadequate. When a device causes an interrupt, the processor saves the address of the next instruction in the program currently running and transfers control to an interrupt handler program (sometimes called a device driver) to service the interrupt. Usually data are read from or written to the device before control is returned to the interrupted program. Data are passed between interrupt handlers and tasks using the communications services provided by the system executive. Some system executives provide I/O device drivers; others require the user to write them. The advantage of user-written drivers is that they may be efficiently tailored to the specific needs of the application.

A clearly defined, efficient system executive provides a "skeleton" upon which an application program may be built. It separates program control and communication functions from application-dependent operations. Thus, the two aspects of a real-time program may be designed and maintained independently. A well-designed system executive should require few changes over the life of a program. By contrast, application tasks will probably undergo considerable change. This separation reduces the possibility of application changes obscuring or disrupting the basic structures of the program as a whole.

The designer has a number of options available in selecting a system executive. Standard executives may be purchased in the form of a software package or a firmware ROM chip. The primary advantages of this approach are the possibility of immediate cost savings and the elimination of in-house program development delays. A possible disadvantage is the designer's lack of knowledge and control over the operations performed within the executive. A standard executive must be designed to satisfy the requirements of a large class of applications.

The alternative approach is to define and write a custom executive tailored to the needs of the specific application. In this way the designer can optimize executive performance as the project evolves. This approach makes sense for systems in which timing constraints are either not well understood or else cannot be predicted in the early stages of the project. It is also a good choice for systems that are likely to undergo substantial modification over their lifespans.

The prospect of having to design and implement a complete system executive can be unsettling to a project leader who is faced with a deadline and limited resources. He must assess the size of the effort involved before the make/buy decision is made. The minimal system executive described here has been successfully implemented; hence it may serve as the basis for the specification of an executive program package. The material presented here may also be used to estimate the effort required to produce this component of a real-time program.

The system executive consists of two major components: a task scheduler and a set of message queue access functions. The scheduler initiates application tasks on a priority basis in response to wakeups from interrupt-handlers and other tasks. Messages are passed between tasks and between interrupt-handlers and tasks by means of circular queues. The queue-access functions provide standardized access to these queues.

This particular implementation uses the C language and a target Z80 microprocessor so that specific design details can be discussed. However, the concepts presented should be applicable to systems using other languages and/or processors. The C language has been found to be particularly suited to the needs of real-time systems. It might be considered to be an intermediate level language that combines the expressive power of a high-level language with the ability to access memory locations directly, a characteristic of assembly language. The particular C compiler used also provides for in-line machine code for situations that require direct control of hardware resources. With both the system executive and the application functions written in C, the entire microprocessor program may be linked into one module. This simplifies the interface between the two program components.

Scheduler

Task-scheduler designs may be grouped into two general categories: preemptive and non-preemptive. A task running under a preemptive scheduler may be suspended if a task of higher priority is awakened by an interrupt handler. This is important for systems in which data received by an interrupt handler must be processed immediately by a task awakened by that handler. Preemptive scheduling may also be necessary if the data rate is such that there is the possibility of data being overwritten before it can be processed. This problem can sometimes be solved by double-buffering the data in either hardware or software.

By contrast, a task running under a nonpreemptive scheduler may not be preempted (that is, suspended) even though a task of higher priority is awakened. Except for interrupt servicing, the running task has control of the computer until it suspends itself. In systems using nonpreemptive scheduling it is necessary for tasks to cooperate in using processor bandwidth by limiting the amount of processing performed between voluntary suspensions. This is not a serious limitation in many real-time applications in which timing requirements are not critical. A significant advantage is that it considerably simplifies the scheduler design and therefore reduces memory and execution-time requirements.

Another major advantage of nonpreemptive scheduling is that it reduces the possibility of inconsistent data being passed between program components. Presumably, a task will complete the processing of an entire message before voluntarily suspending itself. By constrast, a task running under a preemptive scheduler may have processed part of the data in a message when an interrupt handler causes a higher priority task to run. This task might change the content of the message that was being processed by the

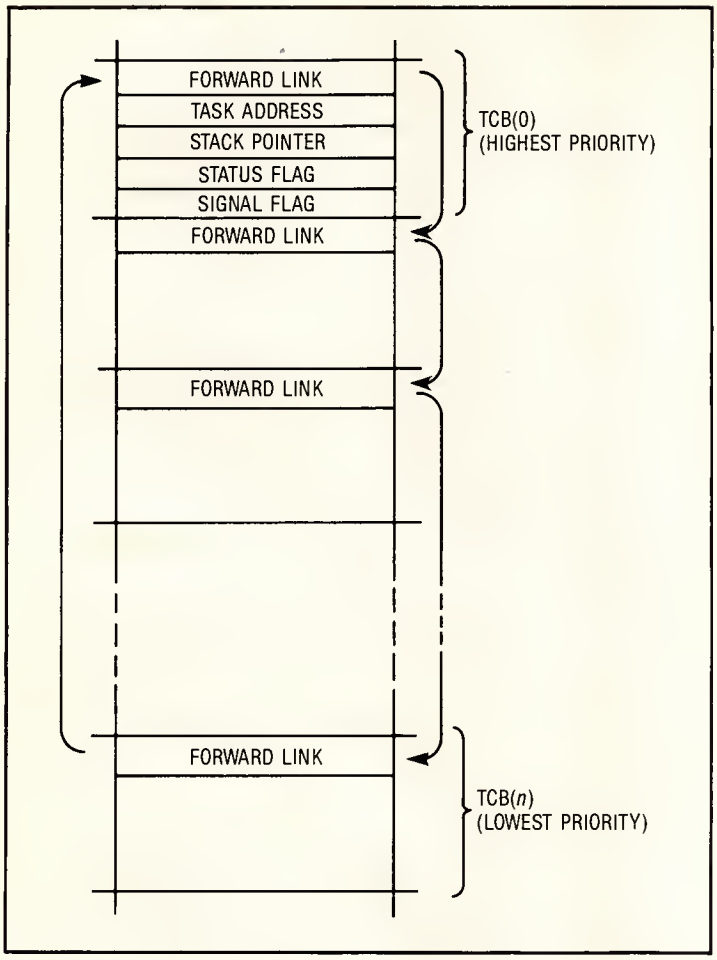


Figure 1. Task control block data structure.

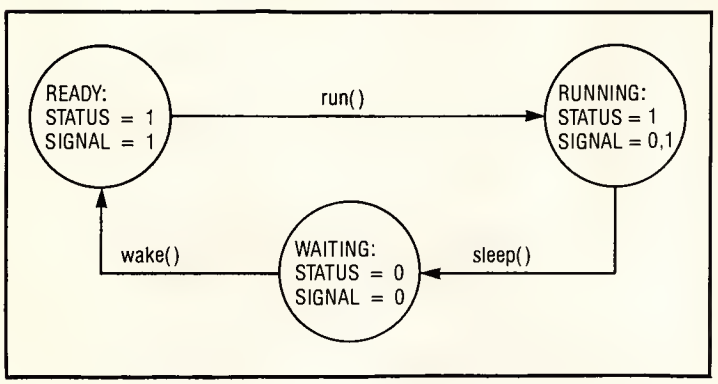


Figure 2. Task Status.

original task. A data access lockout mechanism must be implemented to avoid this problem.*

An objective of this design was to keep the scheduler as simple as possible so that executive execution-time overhead would be minimized. In addition, the anticipated applications did not require the immediate processing of data received from interrupt handlers. As a result, a nonpreemptive type scheduler was chosen.

Task control. The status of each task is maintained in a task control block. The TCBs for all application tasks are contained in a linked list data structure, as shown in Figure 1. A TCB contains a forward-link pointer, used by the scheduler to access TCBs, the starting address of the task, the task's current stack-pointer, and "status" and "signal" flags.

A linked-list data structure is used for the TCB data area primarily for the purpose of determining task priority and to facilitate scheduler operations. When a task suspends itself the scheduler always starts checking TCBs at the beginning of the linked list. As a result, the task described by this TCB has highest priority. Scheduler operations are also facilitated by linking the lowest priority TCB to the highest. Then, during periods when no tasks are scheduled, the scheduler simply searches continuously through the TCBs. Note that this will be the normal idle condition for the task scheduler unless a lowest priority idle task is defined. The idle task must pause periodically to allow higher priority tasks to run.

The linked-list data structure for TCBs is also convenient for systems in which tasks install other tasks to run or in situations in which task priorities must be changed dynamically. The forward-link pointers may simply be changed to reorder the list.

A task may be in one of three states: running, waiting, or ready, as shown in Figure 2. Its current state is determined by the values of its "status" and "signal" flags. A task's state may be changed by calling one of the three functions: run(), sleep(), or wake(). Note that the "signal" flag for a RUNNING task may be in one of two states. This means that a RUNNING task may also be in the READY state. This situation can occur when a RUNNING task is interrupted and scheduled to run again. More will be said about this later.

The scheduler and each task maintain their own stack areas. When a task is interrupted or suspended, its context (that is, the processor's registers and flags) is stored on its stack. Similarly, when the scheduler transfers control to a task, the scheduler's context is stored on its stack. The context is restored when control is returned to the task or scheduler. When a high-level language is used, the scheduler may use the stack area originally allocated by the compiler; task stacks must be explicitly declared as data areas in the program.

When a task is invoked, the scheduler's stack pointer is saved in a memory location. The task's stack pointer is then read from its TCB and loaded into the Z80's SP register. When a task is suspended, the reverse operation is performed. On the other hand, when an interrupt occurs, the context of the running program (task or scheduler) is saved on its stack, but the interrupt handler uses the running program's stack for its operations. It must, of course, pop off all data that it pushes onto this stack before returning to the interrupted program. Interrupts are disabled while the handler is saving and restoring the interrupted program's context. However, they may be enabled while it is performing other operations, since subsequent

*The use of queues to pass data between program components also reduces the possibility of inconsistent data, since new data do not overwrite old data until the space has been released.

interrupt handlers will merely stack the context of the handler they interrupted.

A task program has the general structure shown in Figure 3. It is a C function containing an "infinite" loop. During program startup each task is run from its beginning to the point where it first suspends itself by calling sleep(). During this time the task may perform any task-specific initial setup operations. This would include the initialization of any data items that do not have to be reinitialized during a restart (restart is performed by an initialization task which will be described later). After startup, task entry and exit operations are performed entirely within sleep().

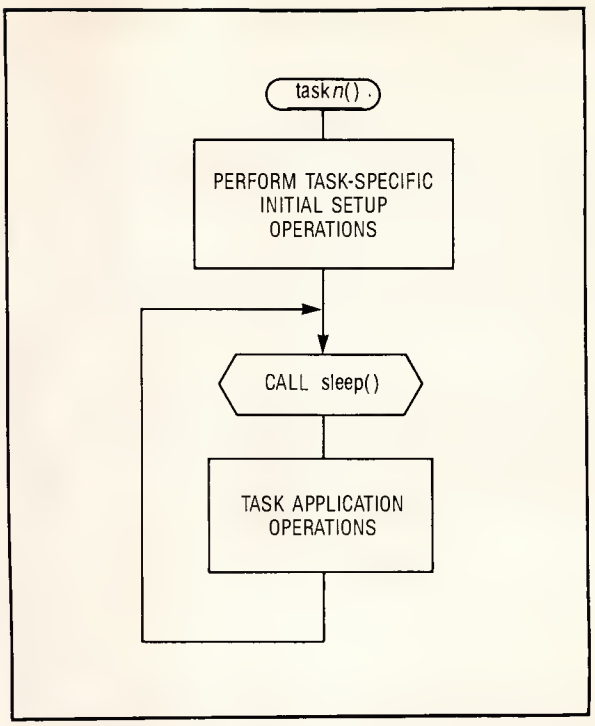


Figure 3. Task program structure.

Scheduler functional components. The scheduler is composed of four basic functions: sched(), wake(), run(), and sleep(). Their relationships to task states are shown in Figure 4. Each is a C function; some contain machine code.

After initialization, the scheduler scans TCBs until it finds one with a "status" flag set. It then clears the corresponding "signal" flag and calls run(); run() transfers control to the previously suspended task and resumes task operation at the point where suspension occurred.

A task suspends itself by calling sleep(). This function clears the task's "status" flag and transfers control back

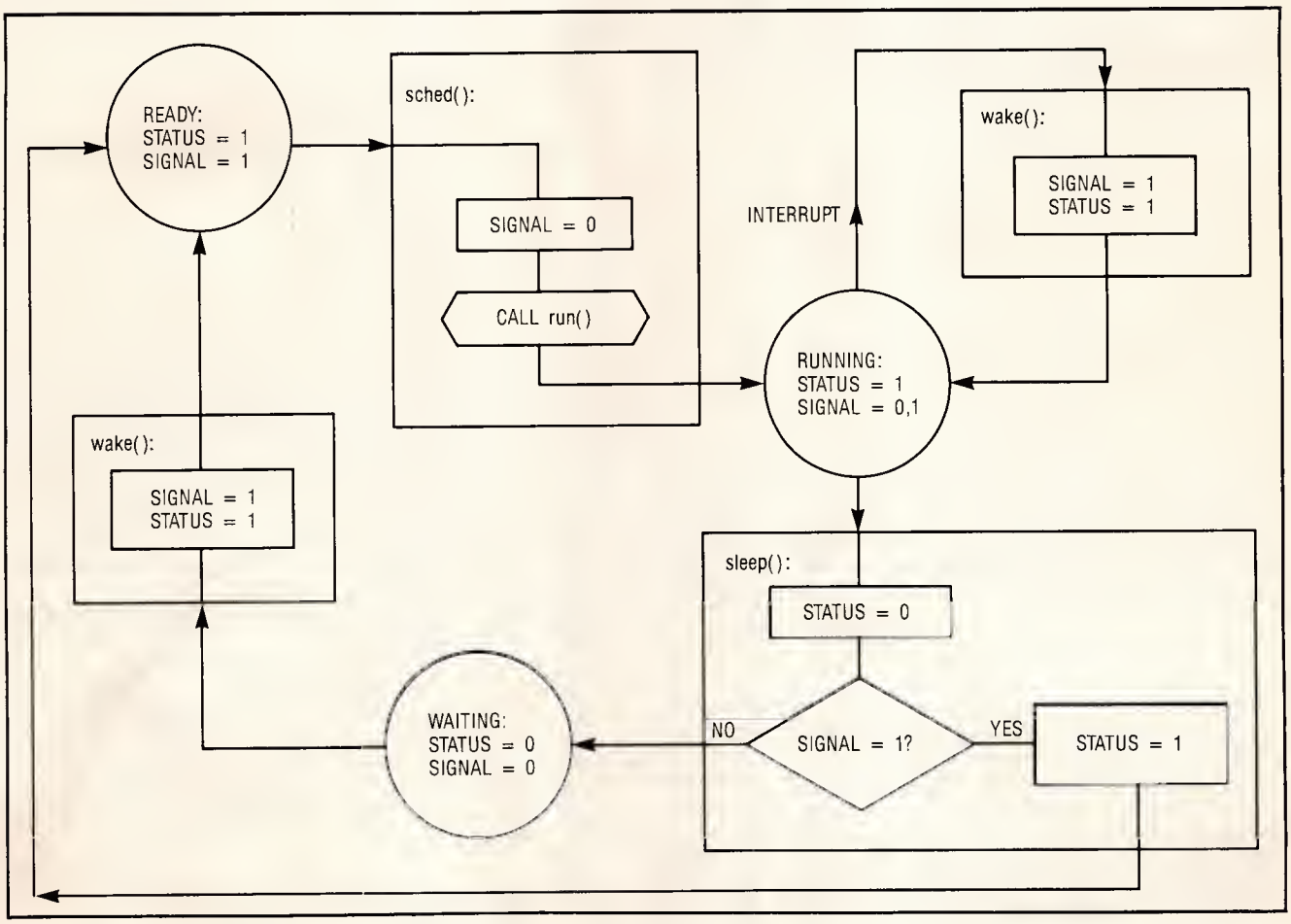


Figure 4. Task states and state change mechanisms.

Language features

Features of C referred to in the main article will be briefly described here for those unfamiliar with the language. Readers interested in a complete description of the language should refer to Kernighan and Ritchie.*

Structured code. C is a structured language. That is, the programmer is provided with a set of programming "constructs" from which to build a program. For example, the "for" statement

```
for ((i = 0; i < 10; i + +){. . .}
```

will perform all statements within the braces 10 times. Index i is initialized before the statements are executed the first time; i is incremented and tested after each execution.

An "infinite" loop may be programmed as follows:

```
for( ; ; ){
    sleep( )  ;
    .
    .
}
```

The statements within braces will be performed "forever." The example represents the basic structure of an application task. Entry and exit from the loop are actually performed within the sleep() function.

Structured data. C provides certain predefined data types, such as "int" for words (16 bits) and "char" for bytes (8 bits). It also provides mechanisms for the declaration of application-specific data types and structures. The following "typedef" structure declares a data type "QUE":

```
typedef struct{
        int head            ;
        int tail            ;
        int length          ;
        char task           ;
        char *pbuf          ;
}QUE
```

A variable may then be declared to have type QUE:

```
QUE timkey                  ;
```

Physically, timkey contains 9 bytes: 6 for the three "int" variables, 1 for the "char task" variable, and two for "char *pbuf." Notation *pbuf specifies that pbuf is a pointer (16 bits) to a "char" variable. Thus, if pbuf is incremented, it will be advanced by one byte position. It is, in fact, the address of a data item. Thus it may be initialized to point to the first location in array tkbuf[] as follows:

```
timkey.pbuf = &tkbuf[0]     ;
```

This statement shows that a variable within timkey may be referenced using the "." notation. It also illustrates the use of "&" to indicate the address of an item of data.

Recall that a memory location has two number associated with it: its address and its contents. C allows a variable name to be declared to represent either number and a means to reference the other. Thus, if pointer (address) pbuf is declared, then the value (content) of the location pointed to is *pbuf. Alternatively, if data array tkbuf[] is declared, then the address of element zero is &tkbuf[0]. This feature of C gives the programmer direct access to computer memory locations.

The items within structure timkey may be initialized as follows:

```
QUE timkey = {0,0,TKLNGTH,0}  ;
```

where TKLNGTH is defined using

```
#define TKLNGTH          6
```

In this context, #define is called a "macro substitution," which simply means that the compiler will substitute "6" whenever it encounters TKLNGTH in the source program.

For programs consisting of many files in which the QUE data type is used, it is convenient to place the declaration in a separate file. The file may then be included in each application file at compile time if a statement such as

```
#include xxx.h
```

is included, where xxx.h is the file containing QUE.

Finally, if a program in one file wishes to reference variable timkey declared in another file, it may do so if it first declares timkey to be external:

```
extern QUE timkey           ;
```

and if timkey has been defined to be a global variable in another file. The external reference will be resolved when the compiled object files are linked together.

One may also declare a local data structure without using typedef. For example, a task-control block data structure might have the following form:

```
struc{
      int    tsklnk         ;
      int    *tskadr        ;
      int    tsksp          ;
      char   tsksta         ;
      char   tsksig         ;
}tcb[NTSKS]
```

This is an array of "NTSKS" tcb structures.

Function entry/exit protocols. When a C function is called, the compiler must provide a means for passing arguments to the function. Most compilers accomplish this by calling an internal (C library) function-entry subroutine. The subroutine constructs a "frame," or region, on the currently active stack to hold the function arguments. In addition, stack space may also be reserved for certain temporary variables declared within the function body (see the description of re-entrancy). A frame-pointer register is set to point to the frame. The compiler then uses offsets from the pointer to reference data.

Similarly, on exit from a function, the compiler calls a function-exit protocol subroutine, which restores the stack to the condition it was in before the function was called.

It is these entry/exit protocol subroutine calls that must be bypassed on entry and exit from an interrupt handler. If they are not, the context of the interrupted program will be lost before it can be pushed on the stack on entry and restored on exit.

To provide support for high-level languages, some of the new 16-bit microprocessors have implemented these entry/exit protocols in firmware.

* *The C Programming Language*, Brian W. Kernighan and Dennis M. Ritchie, Prentice-Hall, Inc., Englewood Cliffs, NJ 07632.

to the scheduler. However, it first checks the "signal" flag. If the signal flag is set, it means that the RUNNING task was also READY, indicating that an interrupt occurred while the task was running and the interrupt handler rescheduled the running task to run again. The sleep() function handles this situation by resetting the task's state to READY (by setting the "status" flag) and returning control to the scheduler. The scheduler then scans TCBs, starting at the highest priority, until it finds a READY task and invokes it.*

A task may be awakened from the WAITING state by calling wake(). This function's single argument specifies the number of the task to be awakened. It is used as an index into the array of TCBs. The wake() function sets the task's "status" and "signal" flags. These are the only operations performed by wake; it is written entirely in C. The wake() function may be called either from an interrupt handler or from a task.

*The order in which the operations are performed in sleep() is important. For example, it would appear that STATUS could be reset after the SIGNAL test, when the SIGNAL test fails. However, if an interrupt occurs between the test and the reset, then STATUS and SIGNAL will both be set (if the interrupt handler reschedules the running task). Then, when STATUS is reset after returning from the interrupt, the final state will be STATUS = 0, SIGNAL = 1, which is a disallowed state. Programming sleep() as shown will avoid this problem.

One additional scheduler program component is the pause() function. It was not included as a "basic" function since it can be constructed from the wake() and sleep() functions. It provides a means for a task to give processor control back to the scheduler voluntarily but, before doing so, reschedule itself to run again. This gives higher-priority tasks that may have been awakened by interrupt handlers a chance to run before the pausing task resumes. The pause() function may be implemented simply as a C function that calls wake() and then sleep(). The task number needed as an argument in the call to wake() may be supplied either as an argument in the pause() call or, since the number of the currently running task is known to the scheduler, it may be supplied automatically.

Scheduler initialization. All C programs start at the beginning of the function called main(). In this application main() performs initial startup operations that do not have to be performed during a program-controlled restart.

One operation performed by main() is to initialize run() and sleep(), as shown in Figure 5, by computing addresses RUNADR and SLPADR, respectively, These addresses are needed when control is passed between run() and sleep() during invocation and suspension of a task, as will be explained later. Since these addresses will be referenced from outside their respective functions, they

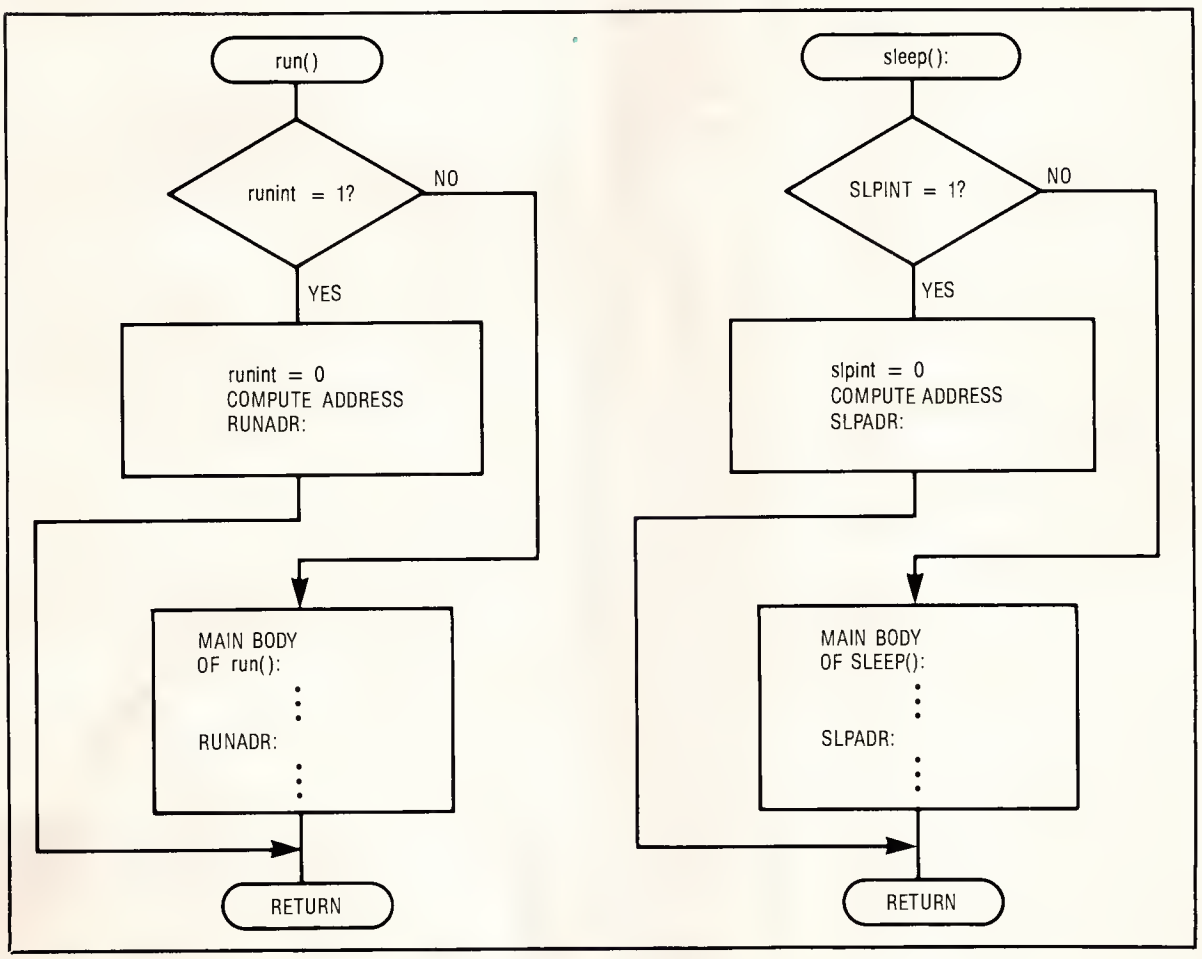


Figure 5. Computing return addresses: RUNADR, SLPADR.

Definitions

These definitions briefly describe term usage within the context of this article. They are not intended to be universal.

Real-time system. This term refers to the response-time of a system to external events. Its use is highly context-dependent. Some systems require response times in microseconds or less to perform properly (e.g., certain feedback control systems). Systems that supply responses comparable to human reaction times (seconds) are also considered real-time. The term is used to distinguish such systems from data processing or computation-intensive systems in which response times are not the central concern.

System executive. A system executive is a collection of functions that provides program initiation and data communication services to application programs. It provides an environment within which a collection of application programs may be operated. Application programs constitute "tasks" which are run by the executive in response to requests from other tasks or external events. In the case of an external event, a signal (or interrupt) is processed by an "interrupt handler" program which posts a request to the executive for a particular task to be run. This action constitutes a "wakeup" or "scheduling" of the task. System executives sometimes provide standard "device drivers" for common I/O devices (e.g., consoles, printers, etc.). These take the place of user-written interrupt handlers and may be much more complex and general purpose.

Application tasks may be assigned priorities. The executive's "scheduler" will then choose the highest priority scheduled task to run next.

Application tasks may pass data, or "messages," by calling executive service routines. These routines maintain message "queues" (sometimes called "mail boxes") to store and forward messages between tasks and between tasks and interrupt handlers (or device drivers).

System executives are sometimes confused with operating systems. An operating system may contain a system executive but is usually much larger and provides many more services, such as program development tools and disk management utilities.

Data structures. Data structures are programming techniques for organizing data so that they may be accessed and updated in an efficient and orderly manner. A variety of data structures have been devised. Each solves a particular data organization problem.

The "queue" data structure is used to control the flow of messages between two asynchronously running programs or tasks. A data queue is analogous to a queue of people waiting to enter a movie. New entries are added at the "tail"; old entries are removed at the "head." However, the queue of people moves forward when "entries" are removed from the head. In a data queue, it would be inefficient to move all the data forward each time an entry is removed. Instead, pointers are maintained indicating the next entry to be removed at the head and the next open entry at the tail. Obviously, as entries are added to the tail, a point is reached when no more room exists in the memory region set aside for the queue. The next entry added must then be "wrapped around" and placed at the beginning of the queue, if sufficient space has been vacated by removal of entries from the head. If the queue is full, the requesting program must be told that the operation has failed. A queue in which entries are wrapped is referred to as a "circular queue."

Another method for passing data between asynchronously running programs is to "buffer" the data. In this method, one or more buffers (blocks of memory) are set aside to hold messages. A scheme is then devised to place new messages in empty buffers and remove messages from full buffers. This method is often used between interrupt handlers and tasks, if the data are received in bursts. The data may be buffered and then processed by the task over a period when the interrupt handler is dormant. Data buffering may be performed in hardware or software, depending on timing requirements.

Another type of data structure is the linked list. This technique is generally used to maintain a list for sequential processing in which the structure of the list may have to be revised dynamically (i.e., during program execution). Entries may have to be added or removed; the list may have to be reordered.

Linked lists are maintained by means of pointers. Each entry contains a pointer (address or index) to the next entry in the list. Sometimes both forward and backward pointers are maintained so that the list may be searched in either direction. Entries in the list may be reordered or removed or new entries may be added by manipulating the pointers. The task control blocks described in this article are organized as a linked list so that they may be searched in a user-specified, priority order.

Processor bandwidth. This term borrows from the vocabulary used in analog circuits. A processor's bandwidth is its instruction execution rate, expressed in instructions per second. When an operation uses too much of the processor's execution time, it is said to be using too much "processor bandwidth."

Access lockout. When more than one asynchronously running program must access a common data region, a mechanism must be provided to ensure exclusive access. Otherwise the data in the region may become corrupted. For example, one task might copy a record (collection of data items) out of a common region, change an item in the record, and write it back. However, without a "lockout mechanism," another task may read the same record, change an item, and write the record back before the first task has completed its operation. The change made by the second task will be overwritten by the record written back in by the first task. The solution is to provide an access "gate" (also called a semaphore) that must be tested to receive access (see the section on re-entrancy for a discussion of semaphores).

Context. The context of a running program is the collection of all data that describe the current state of execution of the program. Physically, in a microprocessor system, it consists of the contents of the internal processor registers, including the program counter, and flags register. This context may be saved on the stack, and the processor may perform some other operation. The context may then be restored and the original program resumed, as if no interruption had occurred.

The term "state" is sometimes used in place of "context." However, in this article, the term is used to describe the status of a task.

Interrupt vector. When a device interrupts a processor, it usually provides an identification byte indicating the interrupt source. This byte is used by the processor to find the address of the appropriate interrupt handler program. The address is called an "interrupt vector."

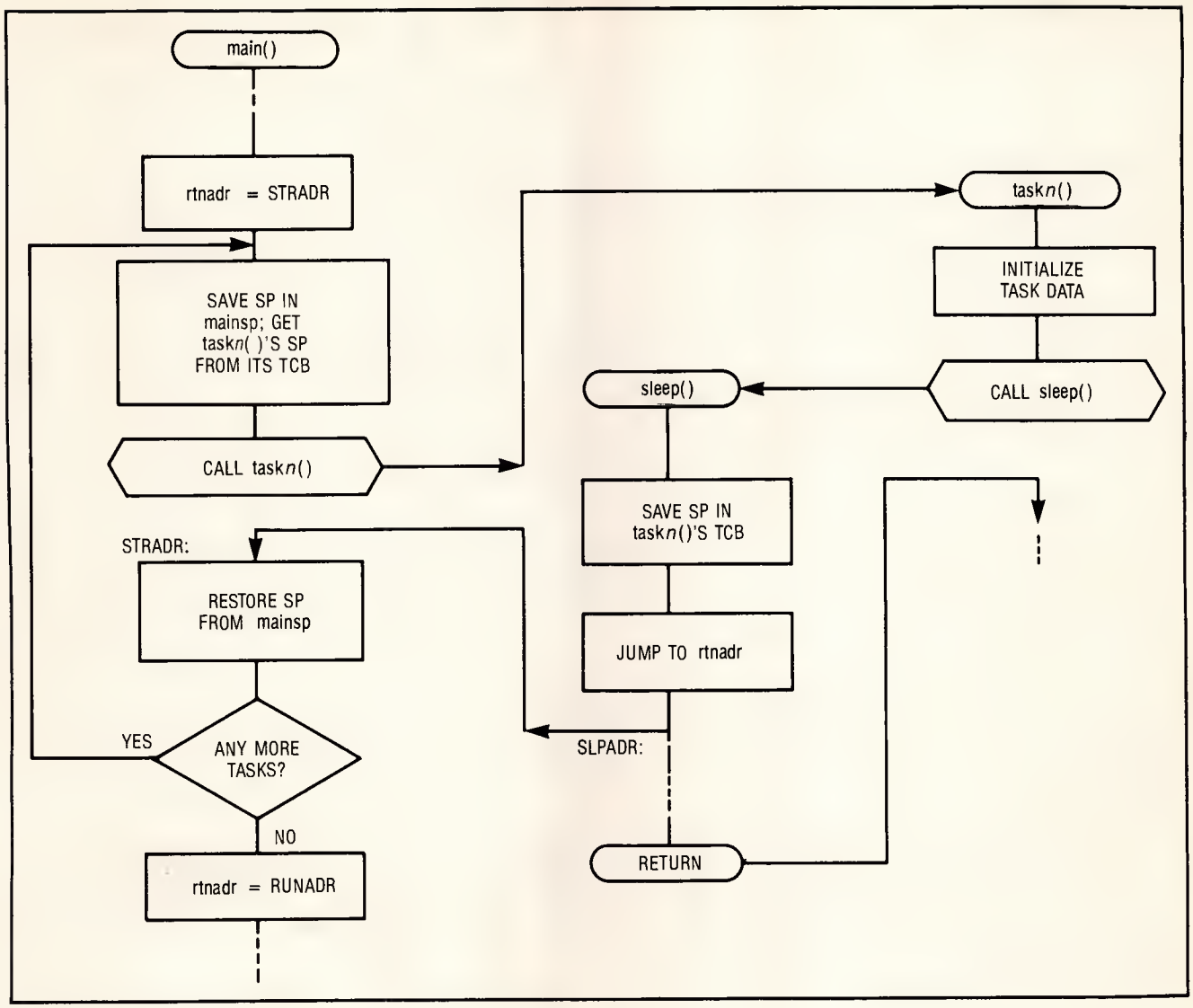


Figure 6. Task initialization.

must be computed and treated as global data items. As mentioned earlier, each task must be called and run to the point at which it first suspends itself. This operation is diagrammed in Figure 6. In main(), return address STRADR is computed and stored in variable "rtnadr." It is used as a return address by sleep() during task initialization operations. Then main() saves its stack pointer and gets the stack pointer for the first application task [in the figure, taskn() represents "task *n*"] from its TCB. It then calls the task using a normal C function call operation. The task initializes startup data, as described earlier, and suspends by calling sleep(). Function sleep() saves the task's stack pointer in its TCB and passes control back to main() via the address in "rtnadr." Function main() then restores its stack pointer and repeats the entire sequence of operations for each application task. At this point sleep()'s return address, "rtnadr," is switched to RUNADR, the entry point in run(). During all future operations sleep() will return to this point.

Scheduler operation. After initialization, scheduler operation proceeds as diagrammed in Figure 7. Function sched() scans TCBs, starting at the beginning of the linked list, until one is found in which the "status" flag is set. It then stores the address of the selected TCB in variable "tcbadr" and calls run(). The run() function saves the scheduler's context by switching to the second set of registers provided by the Z80 (it also pushes registers IX and IY on its stack).* It then saves the current stack pointer and transfers control to address SLPADR in sleep(). The sleep() function gets the stack pointer from the TCB pointed to by "tcbadr" and places it in the Z80's SP register. It then restores the task's context from its stack and returns to the task by means of the normal C

*If the second register bank is needed for some other purpose, the scheduler's entire context may be saved on its stack. For example, the second register bank could be used to save context in an interrupt handler and thereby minimize its execution time.

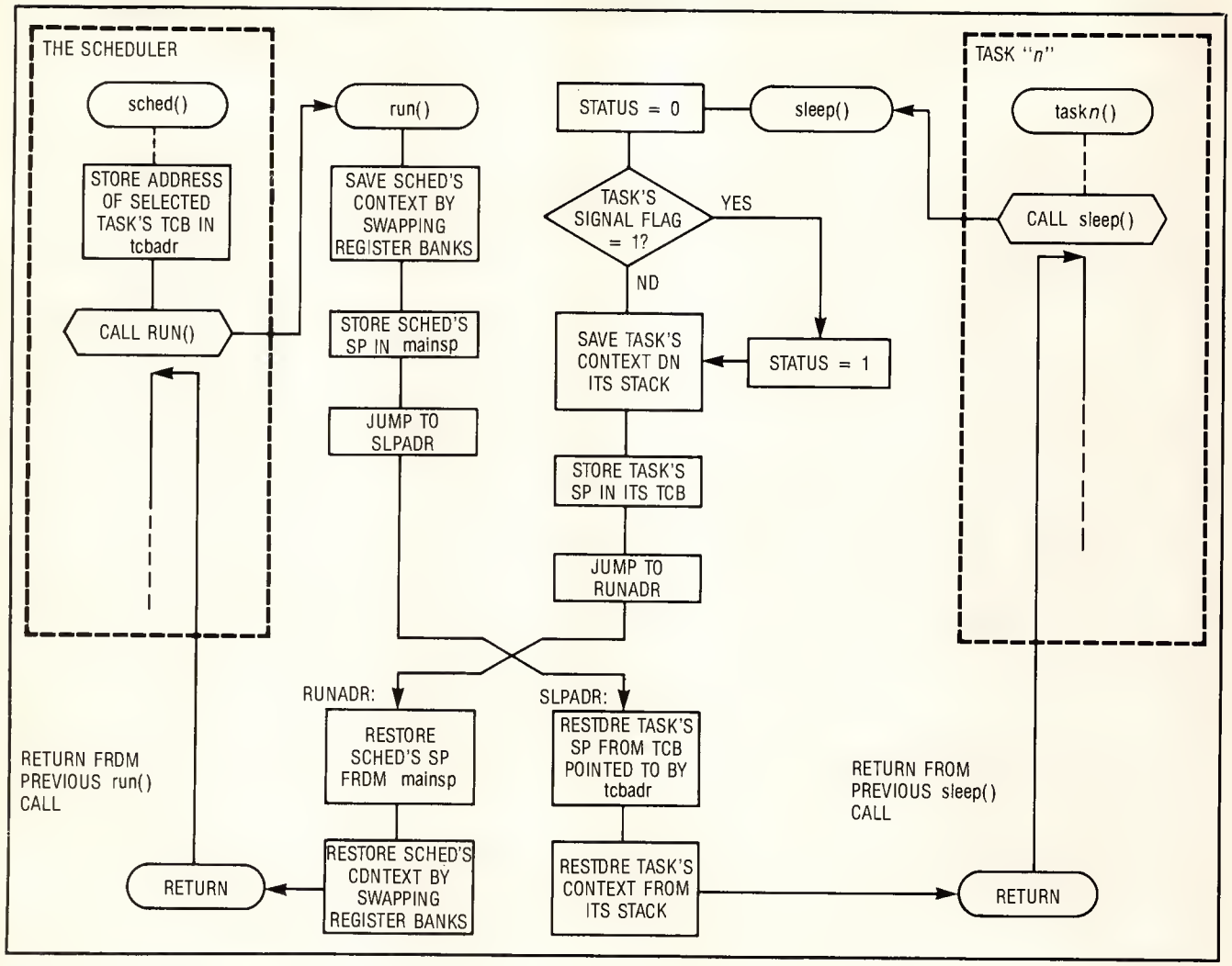


Figure 7. Scheduler operation.

function return protocol. This is possible since the return address was pushed on the stack when the task was initially run to its first call to sleep(), as described earlier.

At this point the task proceeds to perform application operations until it again calls sleep(). As shown in Figure 7, after sleep() clears the "status" flag it checks the "signal" flag. If it is set it proceeds, as decribed earlier, to reschedule the task. In either case it then saves the task's context on its stack, stores the task's stack pointer in its TCB, and transfers control to address RUNADR in run(). The run() function then restores the scheduler's stack pointer and its context and returns to the scheduler via the normal C return protocol. The scheduler then starts at the top of the TCB linked list and scans for another task to run.

Note that as far as the task is concerned, its call to sleep() and the return were just the same as any other C function call. It is not "aware" of the fact that sleep() transferred control back to the scheduler and that other tasks possibly ran before sleep() returned. The same is true of sched() and its call to run(). The scheduler is not "aware" that run() transfers control to a task and receives control back before returning. It is therefore unnecessary for the person writing an application task to know the operational details of the task scheduler; he simply programs a call to sleep() to cause a suspension and expects the task to be awakened at the next C instruction.

Programming interrupt handlers is somewhat more complicated. First, the context of the interrupted function must be saved on the currently active stack. This is performed using PUSH instructions in machine code. After performing I/O operations the context is POPped off the stack and control is returned to the interrupted function.

The interrupt handler is written as a normal C function (with in-line machine code). However, it is not entered or exited using the normal C protocol. Instead, the-interrupt vector is computed so that the first instruction performed is the first PUSH of the context-save sequence. This bypasses the normal C function entry sequence. It is necessary to save the context before any other operations occur so that the interrupted program's context will not be lost. Similarly, control is transferred back to the interrupted program with a normal Z80 return-from-interrupt

instruction just after the context has been POPped from the stack, bypassing the normal C function exit sequence.

An interesting observation can now be made concerning this particular implementation of a task scheduler. It has not been necessary for the implementer to be aware of the details of the C function entry or exit protocols used by the compiler. In interrupt handlers, they are simply bypassed. The running program's context (machine state) is saved, interrupt operations are performed, the context is restored, and control is returned to the running task before the handler's normal C function return operations. Similarly, transfers between the scheduler and tasks are performed within C functions run() and sleep(). These functions save machine context on the currently active stack, transfer control, and restore the context of the destination function from its stack. In this way task scheduling and interrupt operations are transparent to the compiler. The C program's context is saved before these operations are performed and restored afterward.

Queues

The other aspect of real-time program design involves the implementation of a means for passing data between tasks and between interrupt handlers and tasks. Since tasks and interrupt handlers run asynchronously, the use of queues for these operations ensures that messages will not be lost.

Queue design considerations. One must first decide whether the application requires fixed- or variable-entry size queues. Fixed-entry size queues are easier to implement since no wrap-around problem exists at the end of the queue. In addition, the entry size does not have to be included as part of each entry; it can be placed in the queue header. If variable-length queue entries are required, one must decide whether to wrap an entry at the end of the queue or to simply abandon the space at the end and start the new entry at the top. The decision should be based on expected queue entry sizes. This will determine the average amount of space that will be wasted at the end of the queue. The tradeoffs, as usual, are between extra processing time and extra memory space.

Queue structures. Within C it is possible to define a queue header structure data type by using a "typedef" declaration. For a variable entry size queue, this might take the following form:

```
typedef struct{
    int     head        ;
    int     tail        ;
    int     length      ;
    char    task        ;
    char    *pbuf       ;
}QUE
```

This declaration may be placed in an "include[11] file that is attached to each program source file. Entries are added to the queue starting at the "tail" pointer and removed starting at the "head." The queue's length is specified by parameter "length." The parameter "task" will be described in more detail later in the discussion of queue access functions. Briefly, it contains the number of a task that suspended itself when it was not able to complete a requested queue-access operation. Pointer "pbuf" points to the beginning of the actual queue data array. The first byte in each queue entry contains the number of bytes that follow in that entry.

Using the QUE data type it is then possible to define queues and their headers as follows:

```
#define TKLNGTH = 6
QUE timkey = {0,0,TKLNGTH,0} ;
char tkbuf[TKLNGTH] ;
```

This might define, for example, a queue for passing data between a timer task and a keyboard task. The pointer "pbuf" to the queue array tkbuf[] must be initialized to the address of the array with a statement of the following form:

```
timkey.pbuf = &tkbuf[0]        ;
```

This queue may be referenced from the file containing the timer or keyboard task by first declaring the queue header as an external reference:

```
extern QUE timkey              ;
```

and then by referring to its address, &timkey, in, for example, the argument list of a queue access function.

Queue access functions. Various queue-access functions may be written to satisfy different application requirements. These functions may be built upon two primitive functions which will be called putq() and getq().

The putq() function declaration has the form

```
putq(source, dest, count)
```

where "source" is a pointer to an array (or item) of data to be placed in the queue and "dest" is a pointer to the queue header. The "count" function is the length of the message to be moved, in bytes. For example, this function could be used to move a four-byte message from array gmttim[] in the timer task to the keyboard task by writing

```
if (putq(gmttim, &timkey, 4) = = -1)
        sleep( )               ;
```

As indicated, putq() is programmed to return a −1 value if not enough room exists in the specified queue to store a message of length four (including one more byte for the entry size). In the case shown, the programmer has simply chosen to suspend the task if this situation arises.

The function getq() is declared similarly:

```
getq(source, dest)
```

It also returns a −1 if no entry is present in the queue pointed to by argument "source."

Using these two primitive functions to perform the actual queue access operations, one can write other useful

Re-entrancy

The term re-entrancy usually describes a phenomenon that occurs in shared code segments. However, a similar problem occurs in shared data segments.

Code re-entrancy. Consider a situation in which an interrupt handler is part-way through its processing operations when the same interrupt happens again. If the handler re-enables interrupts (after saving the interrupted program's context) then the second interrupt will be serviced. That is, the context of the first interrupt process will be saved and the interrupt handler will be re-entered. The handler will proceed to process the second interrupt's data, perhaps by queueing some input data in an output queue and waking a task. It will then restore the context of the interrupted program and return control to it. In this case the interrupted program is the interrupt handler itself! At this point, two problems may arise. First, if the handler is programmed to store data in fixed memory locations, the data computed when the handler was called the first time have been overwritten by the computations performed by the second call. Second, even if the first problem is solved, the data received by the second interrupt will get queued before the data for the first. The received data will therefore be out of time sequence. The second problem must be solved either by keeping interrupts disabled within the handler or by ensuring that the handler's execution time is less than the minimum time between interrupts.

The first problem illustrates the code re-entrancy phenomenon. It will occur whenever a shared segment of code is called by a number of asynchronously running programs (or by an interrupting device, as described). For example, the queue-access functions, described in the main article, experience the same phenomenon. The problem is solved by ensuring that a separate data segment is reserved each time the shared code segment is entered.

Some modern compilers accomplish re-entrancy automatically by reserving a space (frame) on the stack for parameters passed in a function call and for certain data types defined by the language. A "frame pointer" is used to locate the data on the stack, and a new frame is allocated for each call to the function. Therefore, to write a re-entrant code segment, the programmer must simply use data types for which space will be allocated on the stack by the compiler. In the C language this is accomplished by defining dynamic (i.e., non-static) variables within the body of a function. The compiler will allocate space for them in a stack frame.

Another way to solve the code re-entrancy problem is to have each calling program pass a pointer to a reserved data area as part of the shared segment call procedure. The shared code segment would then use the pointer to locate storage for all internal parameters.

The important point is that each time the shared code segment is called, a separate storage area must be allocated for all variables manipulated by the segment.

Data re-entrancy. Suppose that a system is designed in which data are passed between an interrupt handler and a task by means of a simple buffer, rather than a queue. Suppose further that in a certain instance the task is part-way through processing the data when a second interrupt is received from the same source. The interrupt handler will overwrite the buffer with new data and then allow the task to continue. At this point the task will be working with inconsistent data and the results will be incorrect.

The problem could be described as one of "data re-entrancy." That is, new data were entered into the buffer before the previous data had been completely processed. The same problem can occur for data passed between tasks if the tasks are scheduled preemptively or, in a nonpreemptive system, if the programmer is not careful to complete the processing of a buffer's data before suspending a task.

Note that the use of queues to pass data between asynchronously running program components eliminates the problem. However, if the messages passed are very large, the overhead required by the queue mechanism may be too great and the shared buffer technique may be warranted. In such situations, if certain precautions are observed, the data re-entrancy problem may be avoided.

The problem may be solved by defining a flag called a "gate" (also called a semaphore) for each data buffer. The gate controls access to the shared buffer. If a function wishes to access the buffer it first checks the gate to see if it is open (reset, zero). If it is, it closes it (sets it) and proceeds to manipulate the buffer's data. When it is finished it re-opens the gate so that other functions may access the buffer. The "open" operation may, in fact, wake a task that is waiting for buffer access in a manner similar to that used by the queue access functions getqwt() and putqwt(), as described in the main article.

The only problem in using this technique is that the process of testing the gate and setting it must not be interrupted, since the interrupting program could also test the gate, find it open, and close it. Then two functions would be accessing the buffer simultaneously. To avoid this problem the "test and set" operation must be performed in a single machine instruction. Unfortunately, the Z80 microprocessor does not have this instruction. However, it could be simulated by disabling interrupts while the operation is performed.

A compromise approach might be to use a queue to pass a buffer pointer. Multiple buffers could then be used. If the number of queue entries matches the number of buffers, then the use of putqwt() and getqwt() to pass the pointers would eliminate the overwrite problem if the program is written SD that the pointer must be successfully queued before the corresponding buffer is overwritten.

functions that have general application within a task-scheduling environment and support the orderly flow of messages between tasks. For example, a function may be designed so that when an attempt is made to enter a message into a full queue, the task will be suspended. Later, when a message is removed, the suspended task will be awakened so that it can store its message. The suspended task identifies itself by storing its task number in the queue's header in item "task." Such a function might be declared as follows:

putqwt(source, dest, stask, count)

The function name suggests that the task will put a message in the queue if room exists but will wait [suspend, call sleep()] if not enough room exists. Parameter "stask" specifies the number of the calling task.* Similarly, a function to remove messages from a queue, getqwt(), may be designed so that when a task attempts to remove a message from an empty queue it will be suspended. When a message is later placed in the queue the suspended task will be awakened.

Since only one location in a queue's header is allocated for storing the number of a waiting task, a queue can have only one source task using the putqwt() function and one destination task using the getqwt() function. A more elaborate scheme could be devised in which a queue of waiting tasks could be maintained. However, the implementation of separate queues for each data path avoids the problem and simplifies the design.

By using putqwt() and getqwt() it is possible to control the execution of tasks in response to the availability of messages to process in their input queues and the availability of space to store messages in their output queues. For example, if an output device becomes momentarily blocked, the queues that feed it messages will become "backed up," thereby suspending the corresponding tasks. When the device becomes unblocked, the tasks waiting to send data will be awakened in an orderly manner, based upon the availability of storage space in their respective output queues. The flow of messages is similar to the flow of automobiles in a traffic tie-up on a major highway.

System operation

After the initialization of each task, as described earlier, main() calls wake() to schedule the initialization task, init(), and sets the TCB linked-list pointer to init()'s TCB so that the scheduler will test init()'s TCB first [init() is the highest priority task]. Function main() then calls sched() and, from this point on, control is never passed back to main() unless the system is reset from hardware. The sched() function then checks init()'s TCB and passes control to it.

Task init() performs the initialization operations required first during startup and later during restart. That is, init() is programmed so that if another task or interrupt handler wakes it, it will restart the application program from the beginning. It performs these operations with interrupts disabled, and it enables interrupts just before it suspends itself.

The first operation performed in init() is to link all the TCB's together in a loop. This is accomplished by setting the forward-link pointers in the TCBs. As described earlier, the last (lowest priority) TCB is linked to the first so that the scheduler will loop looking for READY tasks. The init() function then performs other application-specific restart initialization operations and finally suspends itself by calling sleep(). These operations may involve waking application tasks. However, if other tasks are not awakened, the scheduler will simply loop until an interrupt handler wakes a task.

Tasks and interrupt handlers provide the various data processing and control services required by the particular application. For example, most real-time applications will be required to service interrupts from a hardware timer device. The interrupt handler wakes a timer task which, in turn, provides interval timing services to other application tasks. Applications that provide a man-machine interface will usually need to service a keyboard. The interrupt handler for this device places the received keystroke character in an output queue and wakes a keyboard task. This task will then input the character from the queue and process it. Tasks and interupt handlers may also be defined to process data from other sources such as communication channels or measurement sensors.

Tasks processing data inputs will usually wake tasks that provide data to output devices. These might be video or alphanumeric displays, communication channels, or equipment controllers. The typical output device is designed to receive a byte of data and then produce an interrupt when it is ready to receive the next byte. The interrupt handler must therefore be designed to output subsequent bytes until the entire message has been sent. It then wakes the output task. The output task must initiate the transfer by sending the first byte and then suspend itself and wait to be awakened by the interrupt handler. It may be assured that the wakeup was from the interrupt handler if the handler sends a "signal" byte message to the task before waking it. The task, on being awakened, checks the queue from the interrupt handler and possibly other input queues to determine the source of the wakeup.

To date, the executive has been used in six projects. Applications have ranged from a data acquisition unit to a rather complex multiprocessor control/display unit incorporating keypad, color graphics, and audio annunciation devices. The project teams have experienced substantial improvements in software development and subsequent maintenance costs. In spite of diverse system requirements on these projects, the executive has undergone few changes since its initial shakeout period.

*Note that the number of the task currently running is the index of its TCB in the TCB linked list structure. Since this is known to the scheduler, it need not be specified as an argument. This would eliminate a possible source of programming error.

Since the executive's design has been kept simple and has been implemented in a high-level language, it may be understood both by those assigned to write the initial application programs and those assigned to maintain the system. Thus is not beyond the means of a small programming team to construct a software environment within which real-time application programs may be produced at reasonable cost. ■

Acknowledgments

I would like to express my appreciation to Dr. Steve Campbell and Mr. David A. Spencer for providing initial design concepts and for assistance in editing this article. Implementation ideas suggested by Dr. John A. Richardson and Mr. Don Schulsinger are also very much appreciated.

This work was performed under the sponsorship of the Federal Aviation Administration. The views and conclusions contained here are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the United States government.

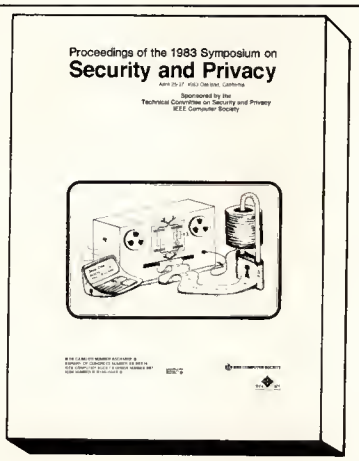
Appendix: Implementation

The executive was originally implemented using a C cross compiler that runs on a minicomputer and produces optimized Z80 code.[1] This provides an excellent environment for producing major real-time systems. However, for more modest systems and to aid the reader interested in the subject of real-time programming, a version has been prepared that uses the less expensive C/80 C compiler[2] This is a "native" compiler, meaning that it runs on the target machine. It produces 8080 microprocessor code that will run on the Z80 machine.

The C/80 version, called CX/80, includes source files for the task scheduler, a set of queue assess functions, example application tasks and interrupt handlers, and two runable demonstration programs. It can be supplied in most 5¼" single sided disk formats. Those interested should contact INTR-SOFT Co., Box 351, Bedford, MA 01730.

1. Z80 C Cross-compiler, Vandata, 17554 Midvale Ave N. Suite 205, Seattle, Wa 98133 (205) 542-7600.

2. C/80 C Compiler, Version 2.0 or 3.0, The Software Toolworks, 15233 Ventura Blvd. Suite 1118, Sherman Oaks, CA 91403 (213) 986-4885.



The proceedings contains 21 papers on topics such as fingerprinting, message authentication, joint encryption and error-correction coding, a distributed secure system, and the denial-of-service problem. 163 pp.

Order #467

PROCEEDINGS—
1983 Symposium on Security and Privacy
April 25-27, 1983

Nonmembers—$28.00
Members—$14.00

Order from IEEE Computer Society Order Dept.
PO Box 80452, Worldway Postal Center
Los Angeles, CA 90080 USA
(714) 821-8380



Walter S. Heath is a Boston-based software consultant. His principal interests are real-time systems software and communication protocols. Previous experience includes the design and implementation of computer navigation, communication, and control systems, including some of the initial computer implementations of Loran and Omega radio navigation systems. This article is based on work he performed for the Federal Aviation Administration at Lincoln Laboratory on the TCAS Airborne Collision Avoidance System and a GPS satellite navigation system. He also contributed to the design of the communication protocol for FAA Air/Ground Data Link Project. Heath received BSEE and MSEE degrees from the University of Michigan, specializing in electromagnetic field theory and computer technology. Before concentrating in real-time computer systems, he performed original research on the application of ferrite materials in the design of microwave antennas and components.

Questions about this article can be directed to the author at 465 Auburn Street, Auburndale, MA 02166: (617) 244-3071.

This proposed standard allows small increments of I/O to be added or modified in an easy, cost-effective way. It is offered here for public comment before submission to the IEEE Standards Board.

P959 I/O Expansion Bus

Proposed Standard

I/O Expansion Bus Subcommittee
IEEE Computer Society Microprocessor Standards Committee

Draft 3 of the proposed IEEE 959 I/O Expansion Bus Standard is the result of an effort by a working group of the IEEE Computer Society's Microprocessor Standards Committee. The P959 bus provides for the conversion of a simple, cost-effective standard input/output bus into a special input/output interface. The final specification will standardize the interface for I/O expansion board products and provide for compatibility throughout the industry. The proposed specification defines the logical, electrical, and mechanical aspects of the P959 bus and P959 I/O expansion boards.

The P959 bus offers a unique design approach to board-level users. To date, board-level designs based on industry standards have dictated that systems be based on single-board computers and I/O boards joined via system backplanes. These backplanes include memory address, control, and data paths. More recently, de facto industry-standard personal computers based on the IBM PC have used this approach to I/O expansion, with the PC bus providing expansion capability to the PC's "motherboard." The expansion boards specified by P959 bring a new concept to I/O expansion—they provide for smaller modules that can be plugged directly onto single-board computers so that the I/O can be tailored to particular applications with minimal cost and space. Expansion modules can convert the general-purpose P959 I/O Expansion Bus into a special-purpose I/O interface such as parallel I/O, serial I/O, RS-232C, and IEEE 488.

The P959 bus signals are derived directly from the on-board address and control buses. The I/O transaction does not require the access arbitration or line buffering that is typical of system-bus I/O; hence, the interface requires less silicon, resulting in lower cost. The mechanical interface between the P959 expansion board and the P959

What is P959?

The P959 bus is a unique interface that facilitates on-board I/O expansion via small, specialized I/O expansion boards which plug directly into base boards. The mechanical interface between a P959 expansion board and a P959 base board is a pin-and-socket connector.

The P959 expansion board concept offers a flexible design approach to board-level users. To date, board-level designers have had to configure systems based on single-board computers and I/O boards joined via system backplane buses. P959 expansion boards bring a new concept to I/O expansion—they provide a family of small modules that can be plugged directly onto single-board computers. With P959 expansion modules, users can tailor their applications directly on single-board computers at minimal cost.

0272-1732/84/0600-0033$01.00 © 1984 IEEE



PROPOSED STANDARD SUBJECT TO REVISION

base board is a unique pin-and-socket bus connector complemented by nonconducting threaded mounting spacers. Consequently, an expansion board plugged into the P959 bus becomes an integral part of the single-board computer.

The P959 committee consists of IEEE members who are employees of single-board computer manufacturers and connector manufacturers as well as of members who are system integrators and end users. Some of the more active members were Leon Adams (chairman) of Texas Instruments, Pete MacWilliams of Intel, Warren Buster of Viking Connector, Richard Main of Signetics, Peter Hoa of National Semiconductor, and Juan Monico of AMD. All were interested in providing flexible, low-cost, single-board computer I/O that would be plug-compatible and interchangeable between different companies' products. Other interested parties, though they were not active committee members, were kept abreast of the committee's activities via mailings of drafts and were encouraged to comment.

The proposed P959 standard started with the iSBX bus, which is currently supported by a number of single-board computer manufacturers. The proposal was put through the wringer for system design, timing, and nomenclature to ensure clarity and compatibility with industry standards. The following is a condensed version of the resulting P959 specification. Portions of Section 4—Mechanical Specification, all of Section 5—Connector Specifications, and Appendices A and B have been omitted from this printing to save space. A copy of the entire P959 specification is available from

Leon Adams, P959 Chairman
Texas Instruments Incorporated
PO Box 1443, M/S 6944
Houston, TX 77001

Readers are encouraged to forward any comments regarding P959—either technical or nontechnical—to Leon Adams at the above address. The committee will discuss such comments and prepare a reply.

The proposed standard

Draft 3, April 11, 1983

Table of contents

1. General

1.1 Scope
1.2 Definitions

2. Functional description

2.1 Bus elements
2.2 Bus signals
2.3 Bus operations

3. Electrical specifications

3.1 General bus considerations
3.2 Timing specifications

P959 committee members

Leon Adams (Chairman)
Texas Intruments

Warren Buster
Viking Connectors

John Craig
Giltspur Microsystems

Peter Hoa
National Semiconductor

Steve Leibson
Consultant

Pete MacWilliams
Intel

Lawrie Eaton
Zendex

Richard Main
Signetics

Juan Monico
AMD

Glen Dokken
Honeywell Information Systems

Douglass Sherrod
Aim Technology

Gordon Force
Force Technology

Tom Thorsteinson
Linear Systems

3.3 Driver and receiver specifications

4. Mechanical specifications

4.1 Connector pin assignments
4.2 Expansion module specifications
4.3* Baseboard specifications
4.4* Board height specifications
4.5* Mounting techniques

5.* Connector specifications

5.1* Reference documents
5.2* Mechanical requirements
5.3* Electrical requirements
5.4* Environmental requirements

6. Levels of compliance

6.1 Variable elements of capability
6.2 Baseboards and expansion modules
6.3 Compliance level notation

List of appendices

Appendix A* Printed-circuit hole size and contact size
Appendix B* I/O connector examples

*To save space, we have omitted Sections 4.3 to 4.5, Sections 5.0 to 5.4, and Appendices A and B. These sections will primarily be of interest to manufacturers—copies of them can be obtained from the working group chairman, Leon Adams, at Texas Instruments Inc., PO Box 1443, M/S 6944, Houston, TX 77001.



PROPOSED STANDARD
SUBJECT TO REVISION

1. General

1.1 Scope

One of the most important parts of any computer system is the I/O. Many systems require different types of I/O depending on the specific application. In addition, the system designer is faced with a significant challenge to keep up with new technology in I/O device types, interfaces, and the LSI components designed to support them. The P959 I/O Expansion Bus is designed to address these needs. Using this standard interface allows small increments of I/O to be added or modified in an easy, cost-effective way.

The P959 I/O Expansion Bus is specified independent of processor or board type. Each expansion interface supports up to sixteen 8-bit I/O ports directly. Enhanced addressing capability is available using slave processors or FIFO devices. In addition, each expansion interface may optionally support a DMA channel capable of data rates up to 2M words/sec. These features are supported for both 8- and 16-bit data paths.

This specification has been prepared for those users who intend to design or evaluate products that will be compatible with the P959 I/O Expansion Bus. For this purpose, functional, electrical, and mechanical specification are covered in detail. The intent of the specification is to guarantee compatibility between baseboards and expansion modules while not restricting the actual designs any more than necessary. Additional design considerations that are not part of the actual specification are presented in the appendices.*

1.2 Definitions

The following paragraphs contain definitions that are used throughout this specification. More detailed definitions can be found in the appropriate section of this specification.

1.2.1 General system term definitions

Compatibility. The degree to which devices may be interconnected and used without modification, when designed as defined in Sections 2, 3, and 4 of this specification. Section 6 discusses the various levels of compliance that the specification allows.

Operation. The process whereby digital signals effect the transfer of data across the interface by means of a sequence of control signals. Operations may be either interlocked or full speed.

Interface. A shared boundary between two systems or parts of systems through which information is transferred.

System. A set of interconnected elements which achieve a given objective through performing a specific function.

Arbitration. The process of determining which requesting device will gain access to a resource.

1.2.2 Signals and paths

Bus. A signal line or set of signal lines used by an interface system to connect a number of devices and to transfer data.

Byte. A group of eight adjacent bits of data that are operated on as a unit.

Word. Two bytes operated on as a unit.

*The appendices are not included with this printing of the draft. See footnote following Table of Contents.



PROPOSED STANDARD SUBJECT TO REVISION

Data path. Signal lines on a bus associated with data.

High state. The more positive voltage level; used to represent one of two logical binary states (see Section 3.1.1 for more details).

Low state. The more negative voltage level; used to represent one of two logical binary states (see Section 3.1.1 for more details).

Signal. The physical representation of a logical value.

Signal level. The relative magnitude of a signal when compared to a reference. Signal levels in this specification are specified in volts.

Signal line. One of a set of signal conductors in an interface system used to transfer messages among interconnected devices.

Active-high signal. A signal for which the logical-true (activated) state is represented by the high electrical state, and the logical-false (deactivated) state is represented by the low electrical state. In this specification, the active-high signal can be identified by the absence of an asterisk (*) at the end of the signal name (refer to Section 3.1.1 for more details).

Active-low signal. A signal for which the logical-false (deactivated) state is represented by the high electrical state, and the logical-true (activated) state is represented by the low electrical state. In this specification, the active-low signal can be identified by the presence of an asterisk (*) at the end of the signal name (refer to section 3.1.1 for more details).

Two-directional signal line. A signal line that may be defined in either direction across an interface, and that cannot be defined in both directions simultaneously. The direction of operation for a two-directional signal line in a system is a configuration option.

Bidirectional signal line. A signal line that may be defined in either direction across an interface. The direction is determined by control signals for each operation.

2. Functional description

The P959 I/O Expansion Bus concept allows low-cost, highly flexible I/O expansion for computer boards. This section provides an overall understanding of the concept by describing the elements of the bus system, the bus interface signals, and the bus operations.

2.1 Bus Elements

There are two basic elements in the P959 I/O Expansion Bus: the baseboard and the expansion module.

2.1.1 Baseboard

A baseboard is any board which provides one or more I/O expansion interfaces (connectors) that meet the electrical and mechanical requirements of this specification. Logically, the baseboard is always the master device, making it responsible for generating all addresses, chip selects, and commands. As a master device for DMA transfers, the baseboard is required to provide the DMA controller function. Mechanically, the baseboard shall supply the latching connector(s) and mounting hole(s) for attaching the expansion module with nylon screws and spacers. See Figures 1 and 2 for mounting details.

2.1.2 Expansion Module

An expansion module is a small, specialized I/O board which attaches to a baseboard. Each expansion module can be one of two standard sizes, single wide or double wide; each is shown in Figures 1 and 2, respectively. The purpose of an expansion module is to convert the general bus interface into a specific I/O interface. An example of this would be an RS-232 serial interface module. The P959 I/O Expansion Bus is specifically designed to simplify the expansion module interface. In many cases, LSI peripheral components can connect to this interface directly.

2.2 Bus Signals

The P959 I/O Expansion Bus signals can be grouped into seven classes based on the functions they perform. The classes are

(1) address and chip select lines,
(2) data lines,
(3) control lines,
(4) interrupt lines,
(5) option lines,
(6) power lines, and
(7) reserved lines.

2.2.1 Address and Chip Select Lines

These lines can be broken into two groups:

Function	Signals
Address	MA0, MA1, MA2
Chip select	MCS0*, MCS1*

The function of these signals is somewhat unique. A typical bus would simply have address lines and require the slave device to create its own chip selects. The P959 I/O Expansion Bus requires the baseboard to generate chip selects. This feature simplifies the expansion modules with only minimal impact to the baseboard.

2.2.1.1 Address lines (MA0, MA1, MA2)

The address lines are active-high signals generated by the baseboard from the low-order address lines. MA0 is the least significant address bit. For an 8-bit baseboard, these lines connect directly to the least significant address

bits A0, A1, and A2. For a 16-bit baseboard, the address line connections are: MA0 to A1, MA1 to A2, and MA2 to A3 (A0 is used along with byte high enable to control byte/word transfers). During DMA operations, the state of the address lines is undefined.

2.2.1.2 Chip selects (MCS0, MCS1*)*

The chip selects are active-low signals generated by the baseboard to enable communication with the expansion module. These lines may transition when they are not required to be valid. It is the responsibility of the expansion module to qualify the chip-select signals with commands (note that this is done internally in many LSI components). Chip selects shall remain high during a DMA operation.

The chip-select lines are defined differently for 8- and 16-bit baseboards as shown in Figure 3. For 8-bit baseboards, each chip select enables eight consecutive ports as specified by the three address lines. For 16-bit boards, there are two cases: an 8-bit expansion module and a 16-bit expansion module.

For a 16-bit baseboard driving an 8-bit expansion module, the chip-select operation is similar to the 8-bit baseboard. The 8-bit expansion module connects to the low byte on the data bus which makes it only accessible through even ports (odd-ports transfer over the high byte). Each chip select therefore enables eight consecutive even ports as determined by the three address lines (note that this mode also allows each chip select signal to enable eight consecutive 16-bit ports for 16-bit expansion modules).

The chip selects on a 16-bit baseboard with a 16-bit expansion module have two functions. In addition to en-

IEEE P959 Draft 3
PROPOSED STANDARD
SUBJECT TO REVISION

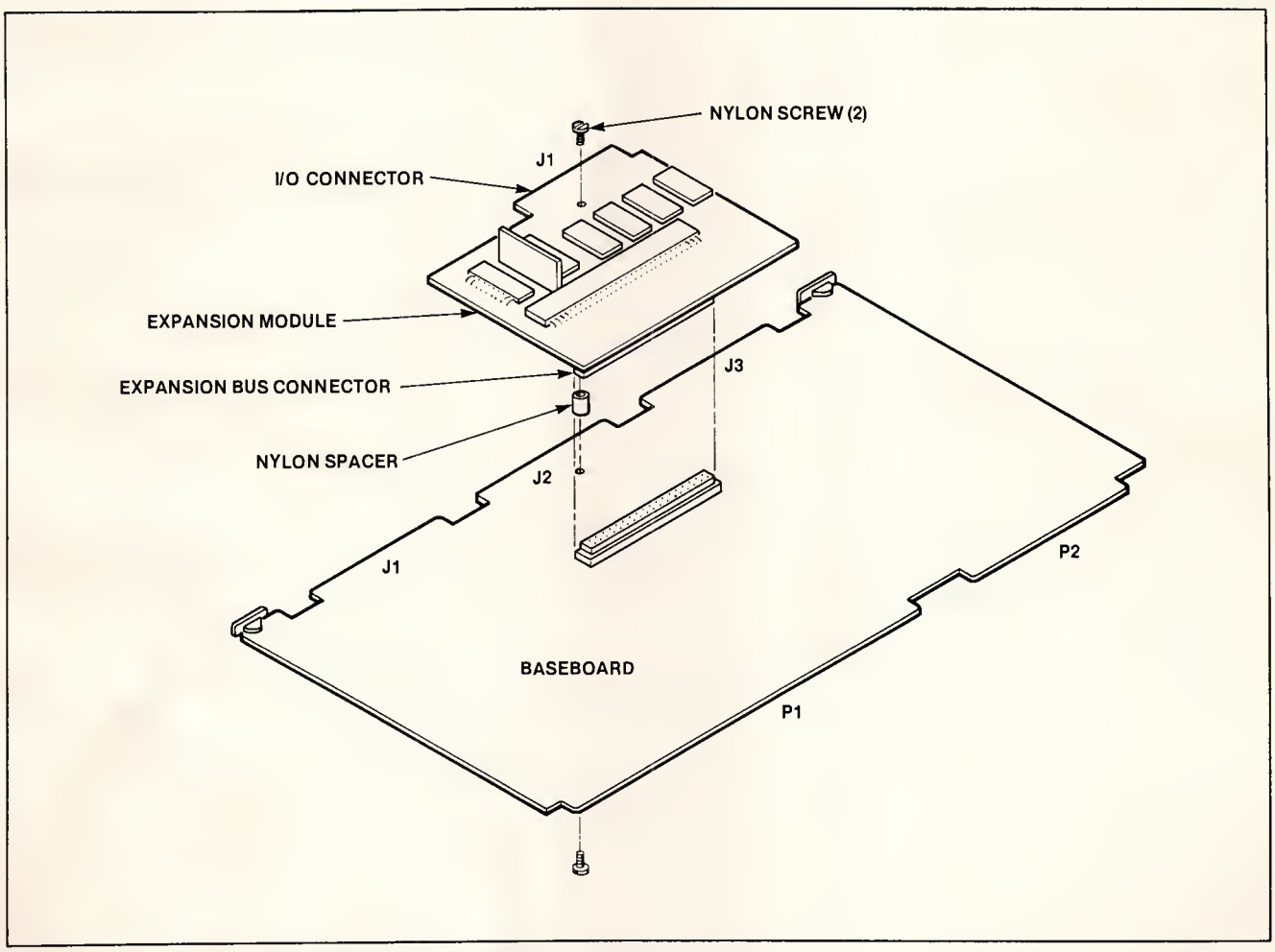


Figure 1. P959 I/O Expansion Bus system with single-wide expansion module.



PROPOSED STANDARD
SUBJECT TO REVISION

abling communication, they control high-byte, low-byte, and word transfer modes. MCS0* is used to enable a low-byte transfer (even ports), MCS1* is used to enable a high-byte transfer (odd ports) and both are used for a word transfer. The three address lines determine which of eight 16-bit ports are being addressed.

2.2.2 Data lines (MD0-MD15)

The data lines are used to transmit or receive data to or from the expansion module ports. The 16 active-high, bidirectional lines are organized in two groups (bytes) of eight lines each. The low byte (MD0-MD7) is used for all transfers on 8-bit baseboards and for all even-byte transfers on 16-bit baseboards. MD0 is the least significant bit of this byte. The high byte (MD8-MD15) is used for all odd-byte transfers on 16-bit baseboards. MD8 is the least significant bit of this byte. Table 1 presents a summary of the data bus functions for different configurations.

2.2.3 Control lines

The following signals are classified as control lines:

Class	Function	Signal
Commands	I/O read	IORD*
	I/O write	IOWRT*
Direct memory access (DMA)	DMA request	MDRQT
	DMA acknowledge	MDACK*
	Terminate DMA	TDMA

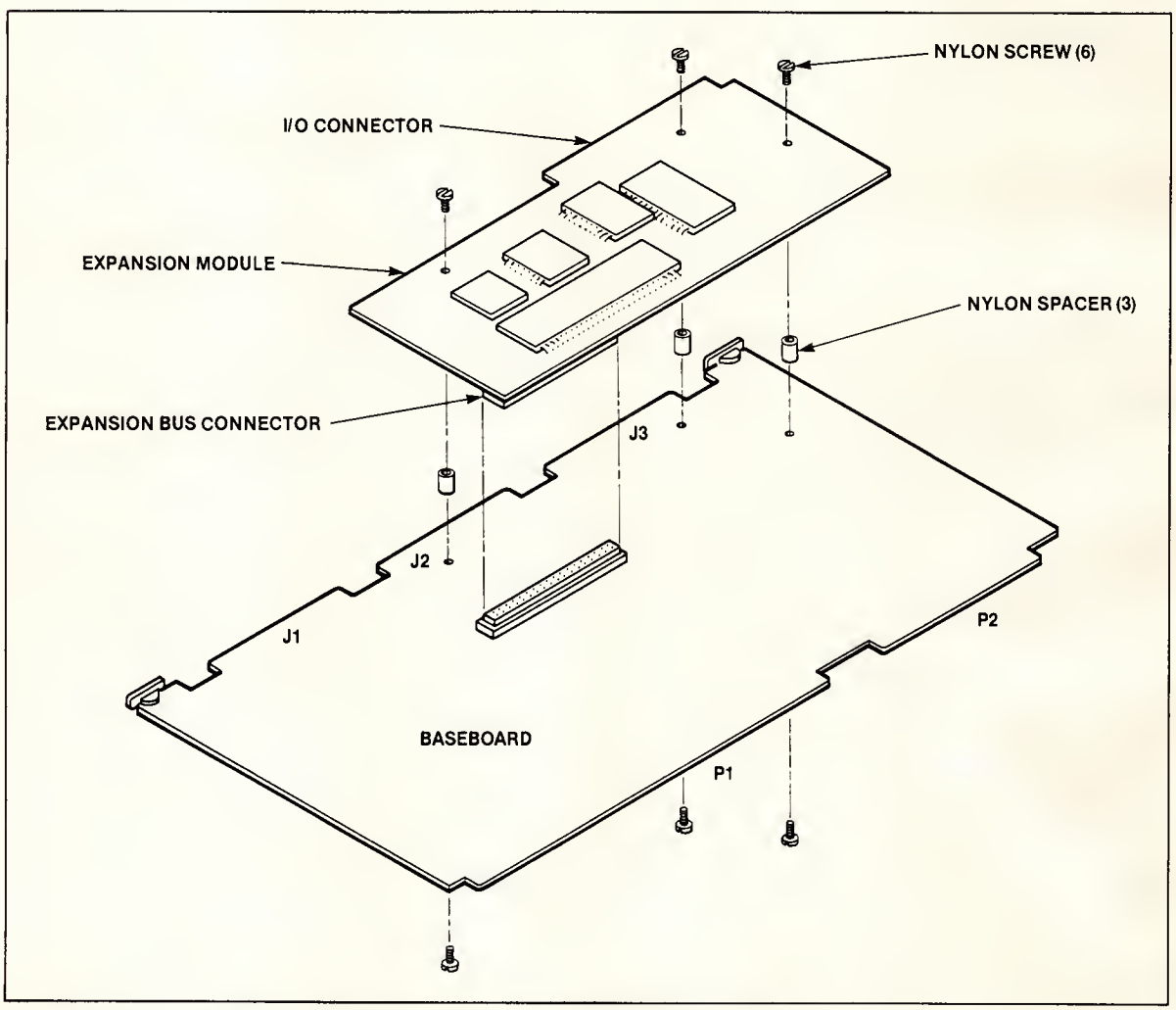


Figure 2. P959 I/O Expansion Bus system with double-wide expansion module.

Initialize	Reset	RESET
Clock	Expansion module clock	MCLK
System control	Expansion module wait	MWAIT*
	Expansion module present	MPST*

2.2.3.1 Command lines (IORD, IOWRT*)*

The command lines are active-low signals generated by the baseboard. An active command line indicates to the expansion board that the address and chip-select lines are valid and that the selected expansion module (i.e., one with active chip select) should perform the specified operation. The I/O read command (IORD*) is used by the baseboard to request data to be transferred from the expansion module. The I/O write command (IOWRT*) is used by the baseboard to transfer data to the expansion module.

2.2.3.2 Direct memory access (DMA) lines

The DMA lines control the communication link between a DMA controller on the baseboard and the expansion module. The use of these lines is optional on both the baseboard and the expansion module.



PROPOSED STANDARD SUBJECT TO REVISION

2.2.3.2.1 DMA request (MDRQT)

This active-high signal from the expansion module to the baseboard DMA controller requests that a DMA cycle be initiated. Upon receipt of this signal the DMA controller begins arbitration for the baseboard's local bus.

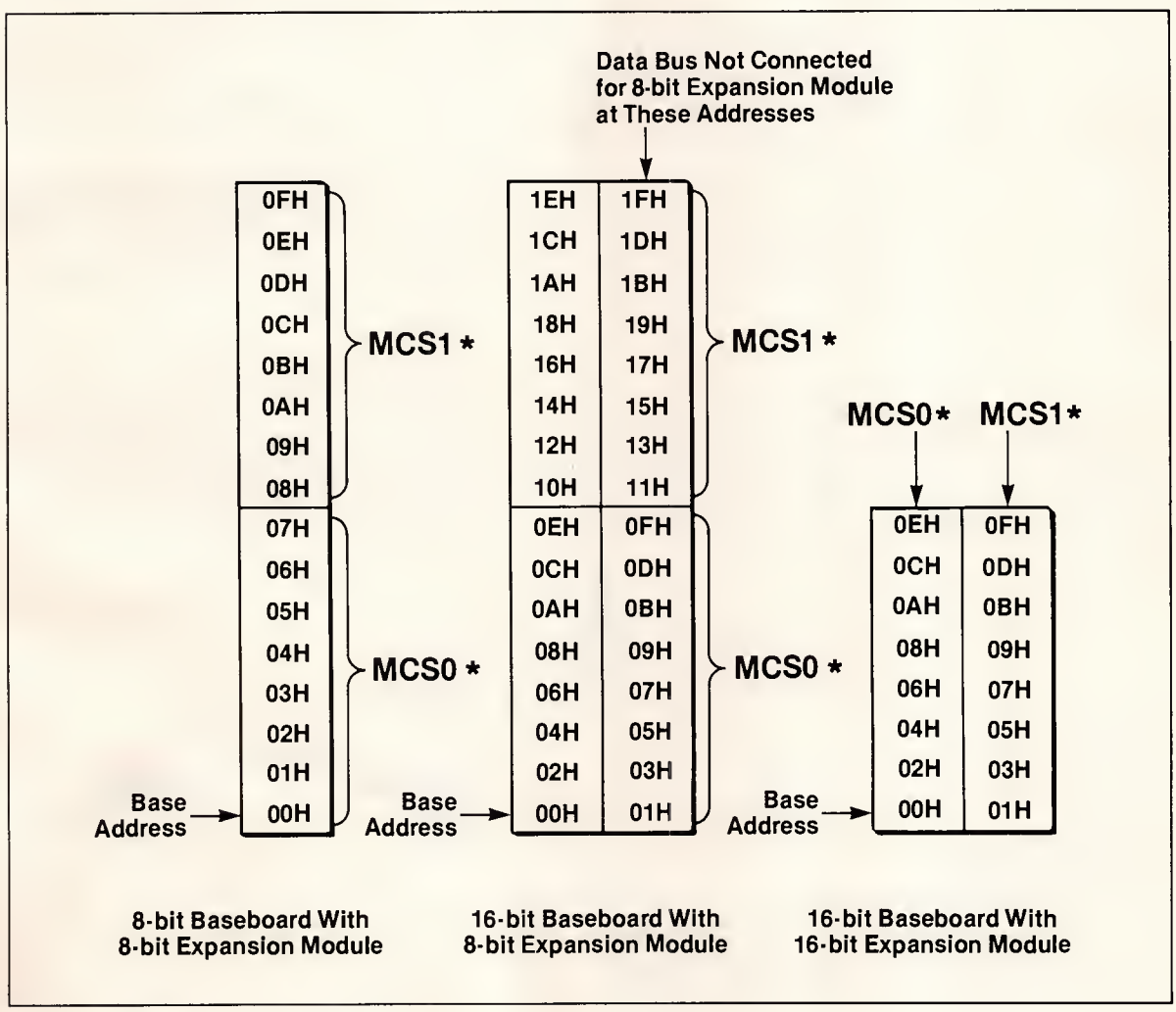


Figure 3. Chip-select address ranges.



PROPOSED STANDARD SUBJECT TO REVISION

2.2.3.2.2 DMA acknowledge (MDACK)*

This active-low signal from the baseboard to the expansion module indicates that a DMA operation has been granted (DMA controller has the bus). Like a chip select, this signal may transition when not valid and therefore shall be qualified by commands on the expansion module.

2.2.3.2.3 Terminate DMA (TDMA)

This is an active-high, two-directional DMA control signal. The specific direction shall be determined by configuration (i.e., jumper or tristate driver). Once configured, TDMA may only operate in one direction. When generated by the expansion module, TDMA is used to terminate the DMA transfer. When generated by the baseboard, TDMA is used to signal the end of a DMA transfer to the expansion module. Baseboards that support DMA shall be configurable to support TDMA in either direction. An expansion module may support or not support TDMA, as required.

2.2.3.3 Initialize (RESET)

This active-high signal is generated by the baseboard to put the expansion module into a known state. The baseboard shall generate a power up reset when power is applied. During normal operation after a power up reset sequence, the expansion module may be reinitialized with a standard reset signal (refer to Section 3.2 for more details).

2.2.3.4 Expansion module clock (MCLK)

This input to the expansion module is a general purpose timing signal. The 9-10 MHz clock is asynchronous to all other bus signals.

2.2.3.5 System control

There are two system control signals (MWAIT* and MPST*) that are defined in the following paragraphs.

2.2.3.5.1 Expansion module wait (MWAIT)*

This active-low signal is generated by the expansion module to extend the current data transfer operation. While MWAIT* is activated, the baseboard is forced to insert wait states into the current bus cycle, allowing the expansion module more time to complete the requested operation. The MWAIT* signal is generated or enabled by a combination of valid chip select(s) and addresses or by a valid DMA acknowledge signal. As these lines change, MWAIT* may transition; however, it shall be stable no later than 75 nsec after MCS0*, MCS1*, and MDACK* become stable.

Recognition of the MWAIT* signal by the baseboard is optional; however, it is strongly recommended. Only in cases where the processor does not support the signal should it be omitted, as it will limit the number of modules that are compatible with the baseboard. Baseboards that do not support the use of the MWAIT* signal may perform only the full-speed operations, as defined in Sections 2.3.1 and 2.3.2. When it supports the MWAIT* signal, the baseboard shall guarantee (usually via a pull-up resistor) that the signal is inactive if not connected. Expansion modules shall support the MWAIT* signal if they cannot meet the full-speed operation specifications.

2.2.3.5.2 Expansion module present (MPST)*

This active-low signal is driven by the expansion module to inform the baseboard that a module is installed. Typically, this signal is connected to ground on the expansion module and to decode logic on the baseboard. When the signal is not activated, the I/O address space normally reserved for the expansion module may be used else-

Table 1.
Data bus function for different configurations.

Configuration	Lines Used	
	MD0-MD7	MD8-MD15
8-bit baseboard	X	—
16-bit baseboard with:		
8-bit expansion module	X	—
16-bit expansion module — even byte	X	—
16-bit expansion module — odd byte	—	X
16-bit expansion module — word	X	X

where. One application of the MPST* signal is to allow an old product that did not include facilities for the P959 I/O expansion to be upgraded with a new baseboard that includes P959 I/O expansion capability. Without expansion modules, the I/O addressing for the new baseboard looks the same as that of the old one. If the new baseboard requires expansion modules, they can be added and their presence recognized by an active MPST* signal.

2.2.4 Interrupt lines (MINTR0, MINTR1)

These active-high lines from the expansion module make interrupt requests to the baseboard. An interrupt line driven from the expansion module shall remain active until the baseboard services it. These lines are asynchronous to all other bus signals.

2.2.5 Option lines (OPT0, OPT1)

These lines are user-defined. Their purpose is to provide connections for any unique system requirements. Examples would be additional interrupt lines or another DMA channel. Baseboards shall connect these lines to wire-wrap posts for configurability.

2.2.6 Power lines

All baseboards shall be capable of supplying +5 Vdc and ± 12 Vdc.

2.2.7 Reserved lines

These two lines are reserved for future enhancements made possible through new technology. Baseboards and the expansion modules shall leave these lines unconnected.

2.3 Bus operations

The P959 I/O Expansion Bus supports I/O read, I/O write, DMA, and interrupt operations.

2.3.1 I/O read operations

There are two types of I/O read operations on the P959 I/O Expansion Bus; the full-speed I/O read and the interlocked I/O read. Whether or not the expansion module



PROPOSED STANDARD SUBJECT TO REVISION

generates the MWAIT* signal determines which type of operation is performed.

Figure 4 shows the full-speed I/O read operation. The baseboard generates a valid address and chip select for the expansion module to initiate the operation. After the set-up times are met, the baseboard activates the IORD* line for a minimum of 300 nsec. The expansion module shall generate valid data from the addressed I/O port within 250 nsec after the IORD* line is activated. The baseboard then reads the data and deactivates the IORD* line. The baseboard is now free to change the address and chip-select lines for the next operation.

Figure 5 shows the interlocked I/O read operation. The baseboard initiates the operation by generating a valid address and chip select, just as in the full-speed I/O read. The expansion module then activates the MWAIT* signal which in turn inhibits the ready signal to the baseboard processor. The baseboard will then activate the IORD* line and insert wait states as long as the MWAIT* signal from the expansion module is active. The expansion module will remove the MWAIT* signal when data is valid on the bus. The baseboard then reads the data and deactivates the IORD* line. The baseboard is now free to change the address and chip-select lines for the next oper-

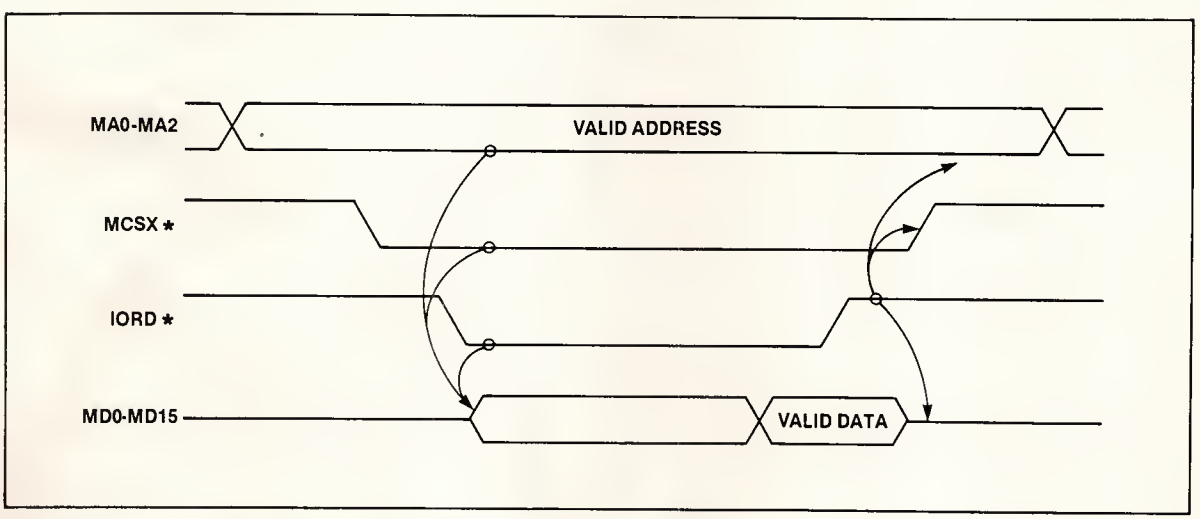


Figure 4. Full-speed I/O read operation.



PROPOSED STANDARD
SUBJECT TO REVISION

ation. The interlocked I/O read operation shall be used by all expansion modules that require a read pulse width greater than 300 nsec or that cannot guarantee valid data on the data bus within 250 nsec after the IORD* line is activated.

2.3.2 I/O write operations

There are two types of I/O write operations on the P959 I/O Expansion Bus: the full-speed I/O write and the interlocked I/O write. Whether or not the expansion module generates the MWAIT* signal determines which type of operation is performed.

Figure 6 shows the full-speed I/O write operation. The baseboard generates a valid address and chip select to initiate the operation. After the set-up times are met, the baseboard activates the IOWRT* line and enables data. The IOWRT* line will remain active for at least 300 nsec and the data will be valid for at least 250 nsec before the IOWRT* line is deactivated. After the IOWRT* line is deactivated, the baseboard is free to change the address and chip-select lines for the next operation.

Figure 7 shows the interlocked I/O write operation. The baseboard initiates the operation by generating a valid address and chip select, just as in the full-speed I/O write. The expansion module then activates the MWAIT* signal, which in turn inhibits the ready signal to the baseboard processor. The baseboard will then activate the IOWRT* line, enable data, and insert wait states as long as the MWAIT* signal is active. The expansion module will remove MWAIT* when it is ready to receive the data. The baseboard then deactivates the IOWRT* line and removes the data. Data shall be stable at least 250 nsec before the IOWRT* signal is deactivated. The baseboard is now free to change the address and chip-select lines for the next operation. The interlocked I/O write operation shall be used by all expansion modules that cannot guarantee proper operation with a 300 nsec write pulse width.

2.3.3 Direct memory access (DMA) operations

DMA is a means of moving a block of data to or from an expansion module without the overhead of an interrupt service or processor polling routine. The source or destination of the data on the baseboard is typically a series of consecutive memory locations. For the expansion module, the source or destination is always an I/O port. The DMA process is always initiated by executing software on the baseboard that sets up the DMA controller and expansion module. This software determines the direction of data movement, the addresses of the source and destination, and the length of the transfer. Once set up, DMA operations will be done automatically by the DMA controller, as requested by the expansion module.

DMA operations are similar to I/O read and I/O write operations. Expansion modules that meet the requirements for full-speed I/O read and I/O write operations can perform full-speed DMA operations. Likewise, expansion modules that meet the requirements for interlocked I/O read and I/O write operations can perform interlocked DMA operations. Once the DMA controller is set up, a DMA operation is initiated by the expansion module activating MDRQT. The DMA controller then acquires (arbitrates) the local bus on the baseboard and activates the

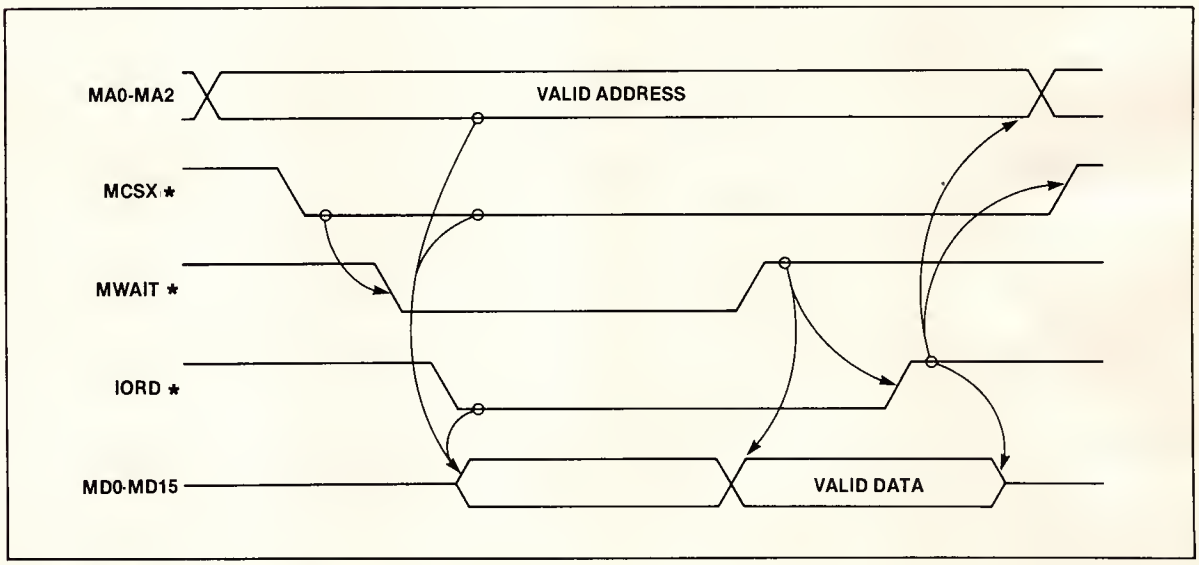


Figure 5. Interlocked I/O read operation.

MDACK* line to the expansion module to acknowledge the request. The MDACK* line functions as a chip select for DMA operations. During a DMA operation, both chip selects (MCS0*, MCS1*) are deactivated and the address lines (MA0-MA2) are undefined. The rest of the operation is the same as the I/O read and I/O write operations described in sections 2.3.1 and 2.3.2, respectively. Figures 8 and 9 show full-speed and interlocked DMA operations, respectively.

DMA operations can be done in two modes: cycle steal and burst. In the cycle steal mode, the expansion module will deactivate the MDRQT line every operation after receiving a command. The DMA controller, in turn, releases the bus after completing the operation. In the burst mode, the expansion module will hold the MDRQT

IEEE P959 Draft 3
PROPOSED STANDARD
SUBJECT TO REVISION

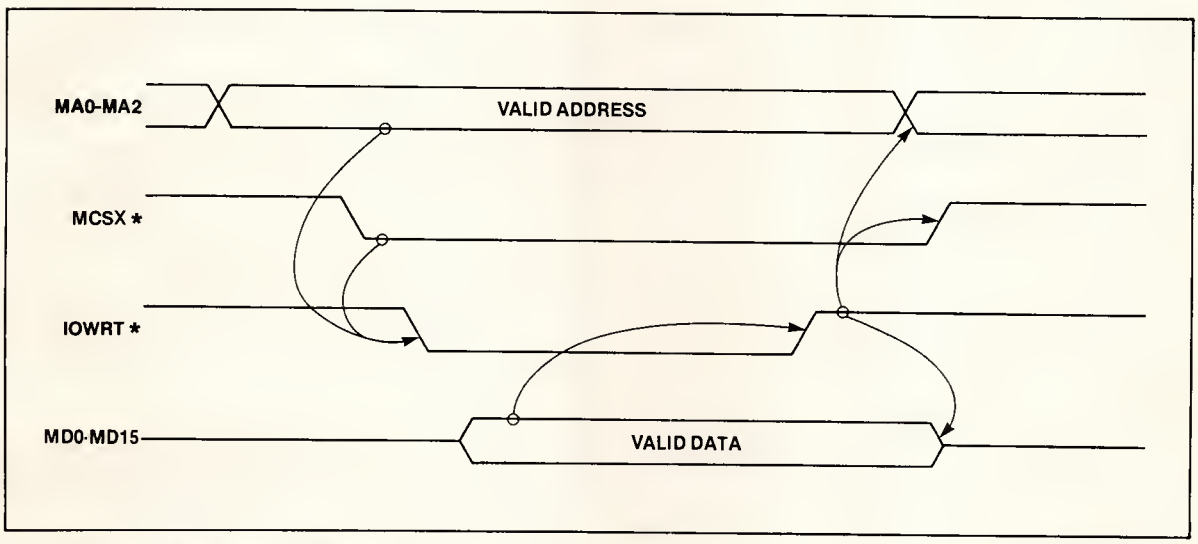


Figure 6. Full-speed I/O write operation.

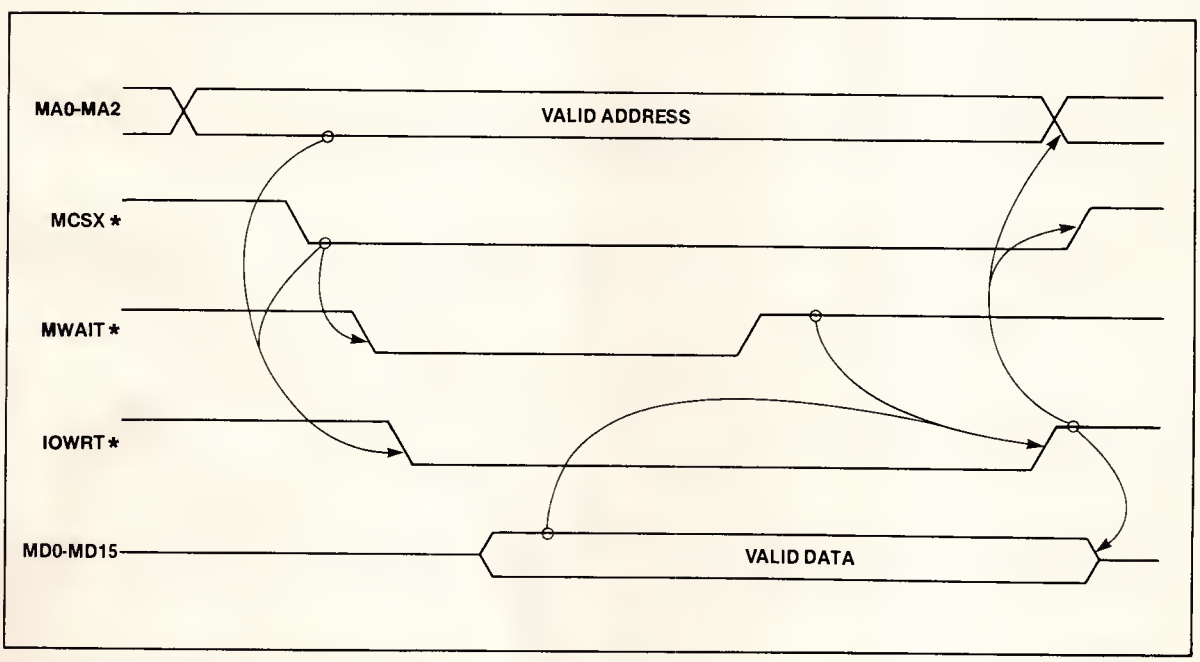


Figure 7. Interlocked I/O write operation.



PROPOSED STANDARD
SUBJECT TO REVISION

line active until the last operation of a transfer. The burst mode will always attain equal or greater throughput, since the DMA controller only arbitrates for the bus once per transfer rather than once per operation.

The TDMA line is provided for additional DMA control. This line is configurable either as an output from the baseboard or as an input to the baseboard.

As an output, the TDMA line identifies the end of a DMA transfer. In this mode, the baseboard activates the TDMA line after the last valid DMA operation is completed. If the expansion module is holding MDRQT active when it receives an active TDMA line, it shall remove the MDRQT line as if it were a command. However, the baseboard need not wait for the MDRQT line to go active before activating the TDMA line.

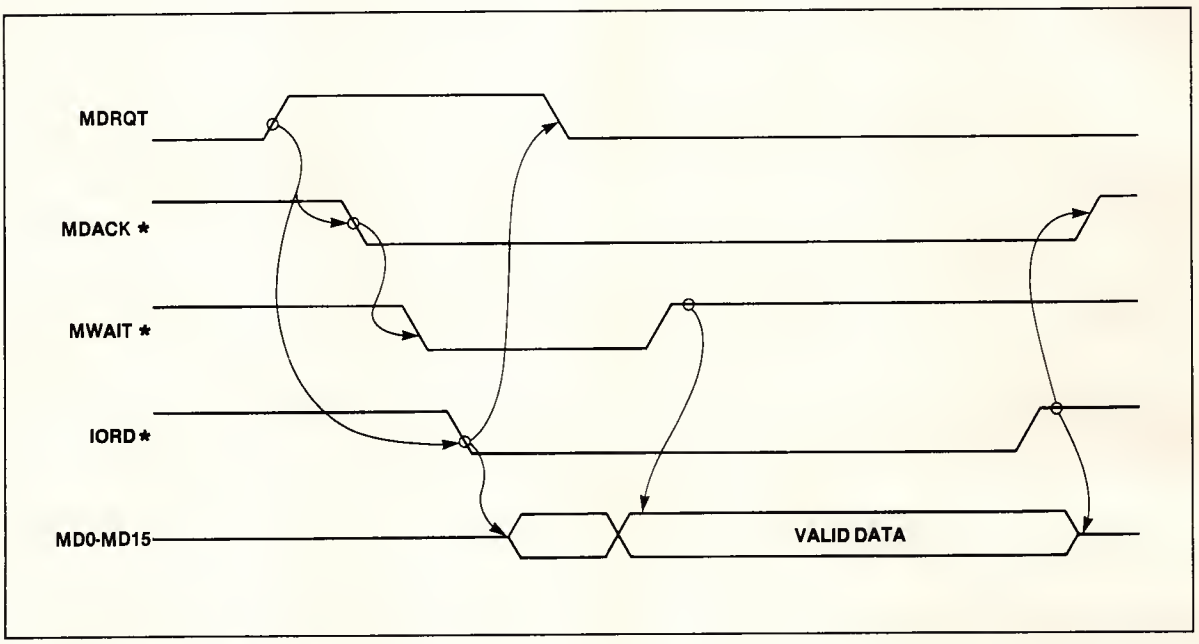


Figure 8. Full-speed DMA operation.

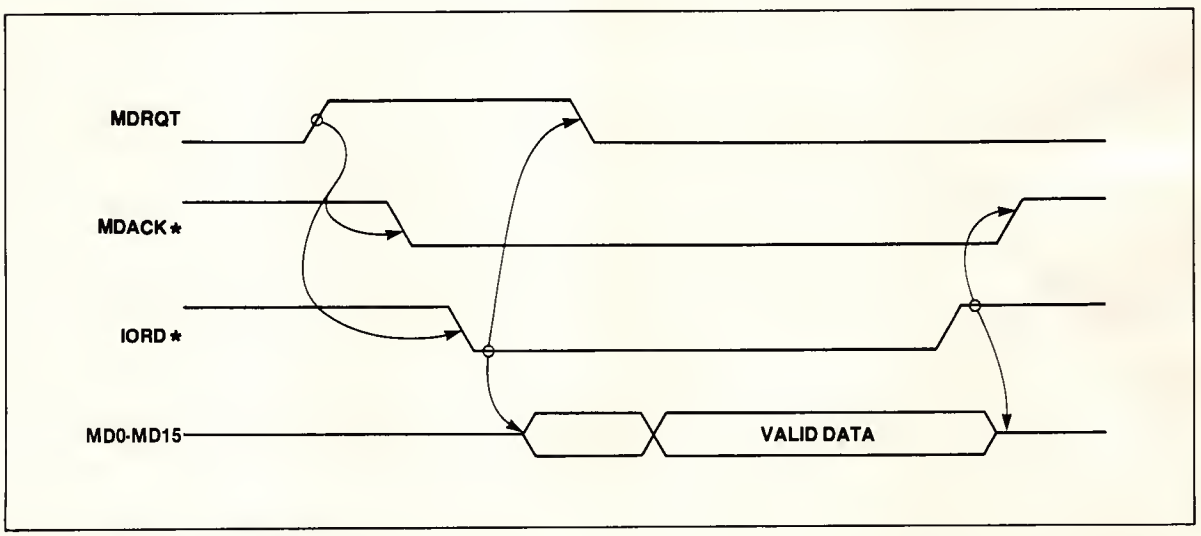


Figure 9. Interlocked DMA operation.

As an input, the TDMA line shall terminate a DMA transfer on the baseboard. To accomplish this, the expansion module first deactivates the MDRQT line, and then activates the TDMA line.

2.3.4 Interrupt operations

The expansion module initiates an interrupt operation by activating one of the interrupt request lines (MINTR0, MINTR1). This signal will interrupt the baseboard processor, causing it to execute an interrupt service routine. The interrupt service routine performs two functions. First, it services the interrupt. This will typically consist of reading data from or writing data to the expansion module. Secondly, the service routine deactivates the interrupt line from the expansion module. In summary, from the expansion module's point of view, the expansion module initiates an interrupt by activating its interrupt signal line and deactivates the interrupt signal line when the baseboard tells it to do so.

3. Electrical specifications

The electrical considerations required for the bus include the following: logical and electrical state relationships, environmental considerations, power supply specifications, signal line characteristics, timing specifications, and driver/receiver specifications.



PROPOSED STANDARD SUBJECT TO REVISION

3.1 General bus considerations

3.1.1 Logical and electrical state relationships

The signal name indicates whether the signal is active-high or active-low. If a signal name ends with an asterisk (*), then the signal is active-low and has the logical-electrical state relationship as listed in Table 2.

If the signal name does not end with an asterisk, then the signal is active-high and has the logical-electrical state relationship as listed in Table 3.

Table 2.
Active-low signal characteristics.

Logical State	Electrical State	Signal Level	
		At Receiver	At Driver
0 (deactivated)	H=TTL High State	5.25V≥H≥2.0V	5.25V≥H≥2.4V
1 (activated)	L=TTL Low State	0.8V≥L≥−0.5V	0.5V≥L≥0V

Table 3.
Active-high signal characteristics.

Logical State	Electrical State	Signal Level	
		At Receiver	At Driver
0 (deactivated)	L=TTL Low State	0.8V≥L≥−0.5V	0.5V≥L≥0V
1 (activated)	H=TTL High State	5.25V≥H≥2.0V	5.25V≥H≥2.4H



PROPOSED STANDARD SUBJECT TO REVISION

These specifications are based on TTL, where the power source is 5V ± 5 percent, referenced to logic ground.

3.1.2 Signal line characteristics

The settling time for commands, interrupts, MDRQT, TDMA, and MCLK, after a transition, is zero. These control lines are used to determine the state of the bus. Ringing beyond the noise immunity levels for these signal lines could cause system failures. Address lines, chip-select lines, data lines, MDACK*, and MWAIT* can ring beyond the noise immunity levels as long as they remain stable for their specified set-up times.

3.1.3 Power supply specifications

Table 4 gives the power supply specifications for the P959 I/O Expansion Bus. The maximum current specification is for one expansion module connector. A baseboard or system may optionally limit the total power to a group of connectors as long as these specifications are met for each individual expansion module connector. The voltage specifications at the connector are measured over the full current specification range.

3.1.4 Environmental considerations

The electrical specifications shall be met within the following environmental conditions.

3.1.4.1 Operating conditions

Bus specifications should be met with operating environmental conditions in the following ranges:

Temperature: 0°C to 55°C (32°F to 131°F) with free-moving air;

Humidity: 0 percent to 95 percent without condensation.

3.1.4.2 Storage

Bus specifications should be met with nonoperating environmental conditions in the following ranges:

Temperature: −40°C to 70°C (−40°F to 158°F);

Humidity: 0 percent to 95 percent without condensation.

3.2 Timing specifications

This section provides all timing specifications on the P959 I/O Expansion Bus. Table 5 summarizes the timing specifications shown in the timing diagrams (Figures 10 through 16). All timing is measured at 0.8 Vdc for low and 2.0 Vdc for high over the full load impedance (resistance and capacitance).

3.3 Driver and receiver specifications

This section specifies all driver types and DC specifications for the P959 I/O Expansion Bus. Table 6 summarizes the specifications.

Table 4.
Power requirements.

Voltage			Current
Minimum Volts	**Nominal Volts**	**Maximum Volts**	**Maximum Amps**
+4.75	+5.00	+5.25	3.0
+11.4	+12.0	+12.6	1.0
−12.6	−12.0	−11.4	1.0
—	GND	—	6.0



PROPOSED STANDARD
SUBJECT TO REVISION

4. Mechanical specifications

This section describes the physical and mechanical specifications that a designer must be concerned with when designing a baseboard or an expansion module compatible with the P959 I/O Expansion Bus. Detailed specifications for the connector are contained in Section 5.*

*Section 5 is not included with this printing of the draft. See footnote following Table of Contents.

Table 5.
AC specifications.

Symbol	Parameter	Min (ns)	Max (ns)	Figure Reference
t1	Address stable before read	50	—	10
t2	Address stable after read	30	—	10
t3	Read pulse width	300	—	10
t4[2]	Data valid from read	0	250	10
t5	Data float after read	0	150	10
t6	Time between commands	—	Note 3	—
t7	Chip select stable before command	25	—	10,11
t8	Chip select stable after command	30	—	10,11
t9	Power up reset pulse width	50 Msec	—	16
t10	Address stable before write	50	—	11
t11	Address stable after write	30	—	11
t12	Write pulse width	300	—	11
t13[2]	Data valid to write	250	—	11
t14	Data valid after write	30	—	11
t15	MCLK cycle	100	110	15
t16	MCLK width	35	t_{15}-35	15
t17[1]	MWAIT* pulse width	0	4 Msec	10,11
t18[5]	Reset pulse width	50 usec	—	16
t19[1]	Chip select or MDACK* to MWAIT* valid	0	75	10,11
t20	MDACK* set up to command	25	—	12
t21	MDACK* hold after command	30	—	12
t22[4]	Command or TDMA to MDRQT removed	—	150	12
t23	TDMA pulse width	300	—	13,14
t24[1]	MWAIT* to valid read data	—	0	10
t25[1]	MWAIT* to write command	0	—	11
t26	MDRQT inactive to TDMA	0	—	14

Note 1: Required only if MWAIT* is activated.
Note 2: If MWAIT* not activated.
Note 3: To be specified by each expansion module.
Note 4: Required in cycle steal mode and for last operation in burst mode.
Note 5: Use t9 for first reset after power up and t18 for any further reset.

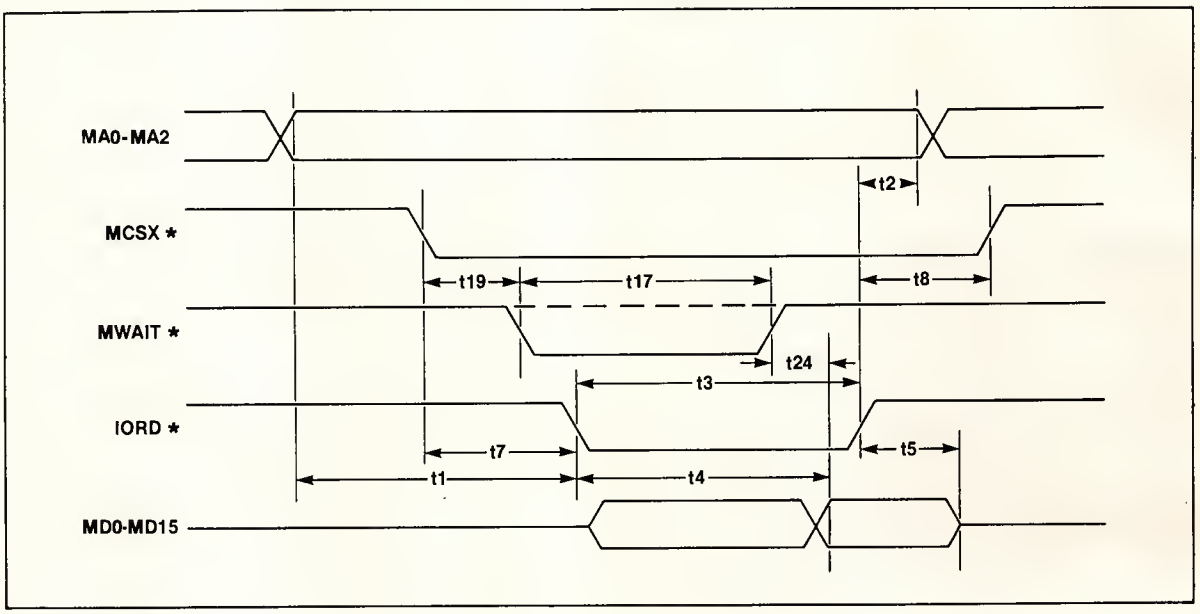


Figure 10. I/O read timing.

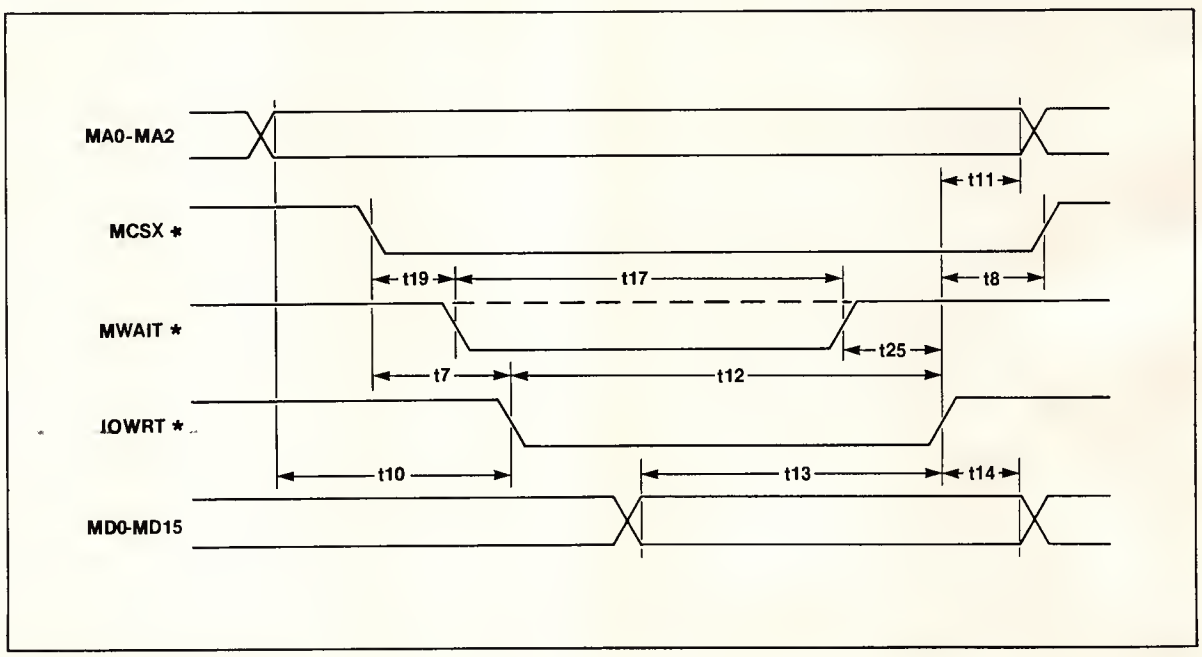


Figure 11. I/O write timing.

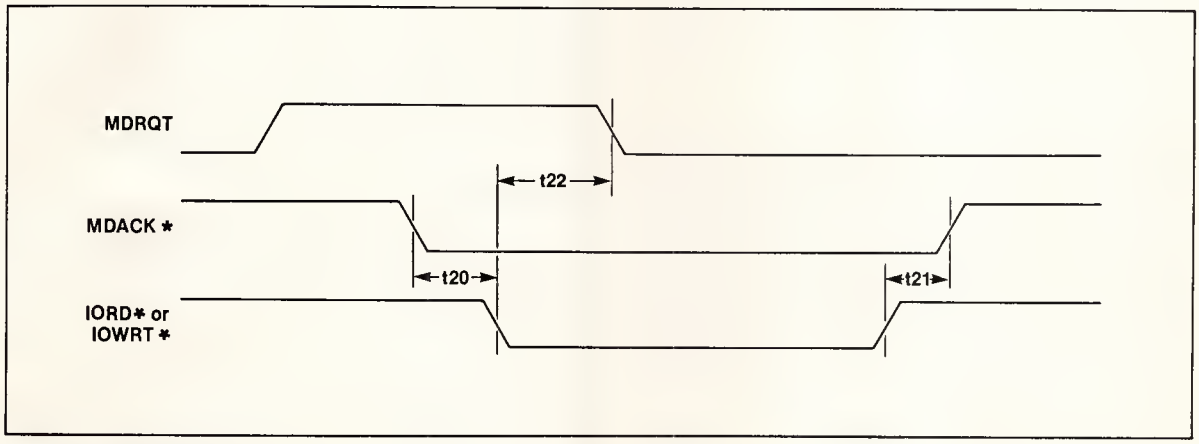


Figure 12. DMA timing.

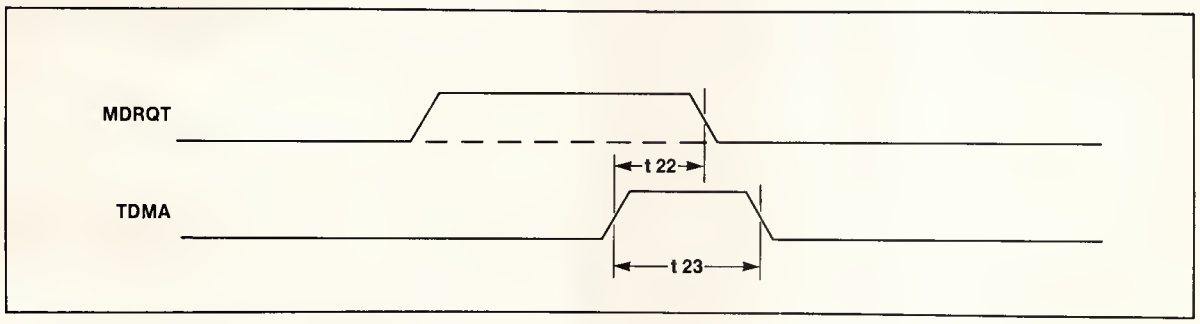


Figure 13. TDMA timing from baseboard.

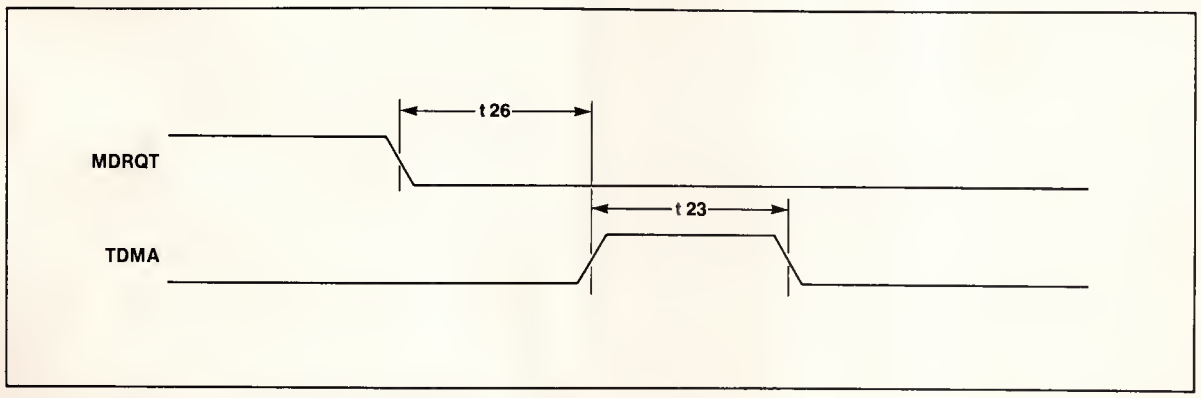


Figure 14. TDMA timing from expansion module.



PROPOSED STANDARD SUBJECT TO REVISION

4.1 Connector pin assignments

There are two types of connector pairs used for the P959 I/O Expansion Bus; a 36-pin, 8-bit version and a 44-pin, 16-bit version. The 36-pin male connector on the expansion module will mate with either the 36 pin (8-bit) or the 44-pin (16-bit) female connectors on the baseboard. The 16-bit, 44-pin male connector on the expansion module will mate with only the 44-pin (16-bit) female connector on the baseboard. The pin locations are shown in Figure 17. Table 7 lists the pin assignments.

4.2 Expansion module specifications

There are two standard outlines for expansion modules. The single-wide outline is shown in Figure 18. The double-wide outline is shown in Figure 19. For each outline, either edge-finger connectors or pin-and-socket connectors may be used. When edge-finger connectors are used, they shall conform to the additional specifications contained in Figure 20. For pin-and-socket connectors, there are no additional restrictions. Details of the I/O connector positioning are determined by the user with some examples provided in Appendix B.*

Sections 4.3-4.5 and Sections 5.0-5.4 are omitted to save space—see note following the Table of Contents.

6. Levels of compliance

This section presents the concept and notation of levels of compliance for the P959 I/O Expansion Bus. The notion of levels of compliance is included to allow the use of P959 I/O Expansion Bus products of varying capabilities manufactured by diverse vendors. It bounds the variability allowed within the P959 I/O Expansion Bus specification, and provides a succinct and convenient notation for these variables.

6.1 Variable elements of capability

The P959 I/O Expansion Bus allows for variation in the data path width, DMA support, and interlocked operation. Each is discussed in the following paragraphs.

6.1.1 Data path

Data path variations exist for both expansion modules and baseboards. Expansion modules may either support 8- or 16-bit data paths. 16-bit baseboards may support only 8-bit data paths or both 8- and 16-bit data paths. 8-bit baseboards may support only 8-bit data paths.

*Appendix B is not included with this printing of the draft. See footnote following Table of Contents.

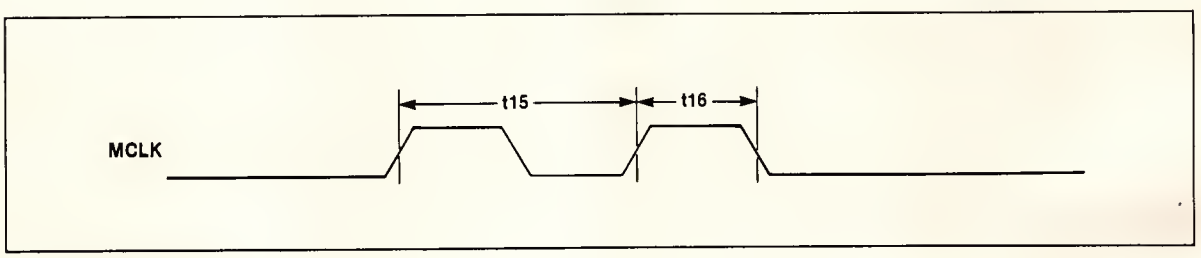


Figure 15. MCLK timing.

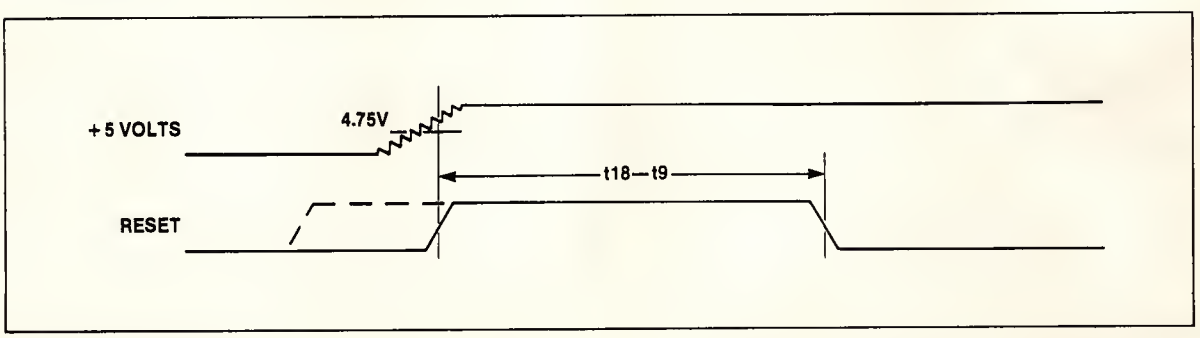


Figure 16. RESET timing.

6.1.2 DMA support

Both expansion modules and baseboards may optionally support DMA operations. In order for DMA to be used in a system, both the expansion module and baseboard shall support the signals that are required (MDRQT, MDACK*, and TDMA).

6.1.3 Interlocked operation

Baseboards may optionally not support this feature. Expansion modules may optionally require this feature. The feature is implemented via the MWAIT* signal. Typically, baseboards will support the interlocked operation and expansion modules will not require it. The purpose for not requiring interlocked operation from all baseboards is to allow the use of low-cost, single-chip microcontroller devices that do not support a ready function.

6.2 Baseboards and expansion modules

When constructing a P959 I/O Expansion Bus system, it is not necessary that all modules have identical capabilities. The only restriction is that the system will only support modes of operation supported by both the baseboard and the expansion module. For example, an expansion module that supports DMA cannot be plugged onto a baseboard that does not support DMA and result in a system that supports DMA. It is the responsibility of the system designer to guarantee that a required system feature is supported by all system components.



PROPOSED STANDARD SUBJECT TO REVISION

6.3 Compliance level notation

The following notation allows a vendor to succinctly and accurately specify a product's level of compliance with the P959 I/O Expansion Bus specification. The lack of an element specification implies that either (1) there is no compatibility or (2) there is no requirement for that element.

6.3.1 Data path

D8 represents an 8-bit expansion module.

Table 6.
DC specifications.

	Driver				Receiver		
Signal	**Type**	I_{OL}	I_{OH}	C_O	I_{IL}	I_{IH}	C_I
MA0-MA2	TTL	—	—	—	−0.5	70	40
MCS0*, MCS1*	TTL	—	—	—	−4.0	100	40
MDACK*	TTL	—	—	—	−1.0	100	40
MD0-MD15	TR1	1.6	−300	130	−0.45	70	40
IORD*, IOWRT*	TTL				−1.0	100	40
MWAIT*	TTL	1.6	−50	40	—	—	—
MDRQT	TTL	1.6	−50	40	—	—	—
TDMA	TTL	1.6	−50	40	−1.0	100	40
RESET	TTL	—	—	—	−2.1	100	40
MCLK	TTL	—	—	—	−2.4	100	40
MINTR0, MINTR1	TTL	2.0	−100	40	—	—	—
OPT0, OPT1[1]	TTL	2.0	−100	40	−2.0	100	40
MPST*	TTL	2.0	−100	40	−2.0	100	40

Note 1: These are recommended specifications and the lines are user-defined; it is the user's responsibility to ensure adequate drive.



PROPOSED STANDARD
SUBJECT TO REVISION

D16 represents a 16-bit expansion module.

D8/8 represents an 8-bit baseboard that can support an 8-bit expansion module.

D16/8 represents a 16-bit baseboard that can support an 8-bit expansion module.

D16/16 represents a 16-bit baseboard that can support an 8- or 16-bit expansion module.

6.3.2 DMA support

DMA represents a baseboard or expansion module that can support DMA operations.

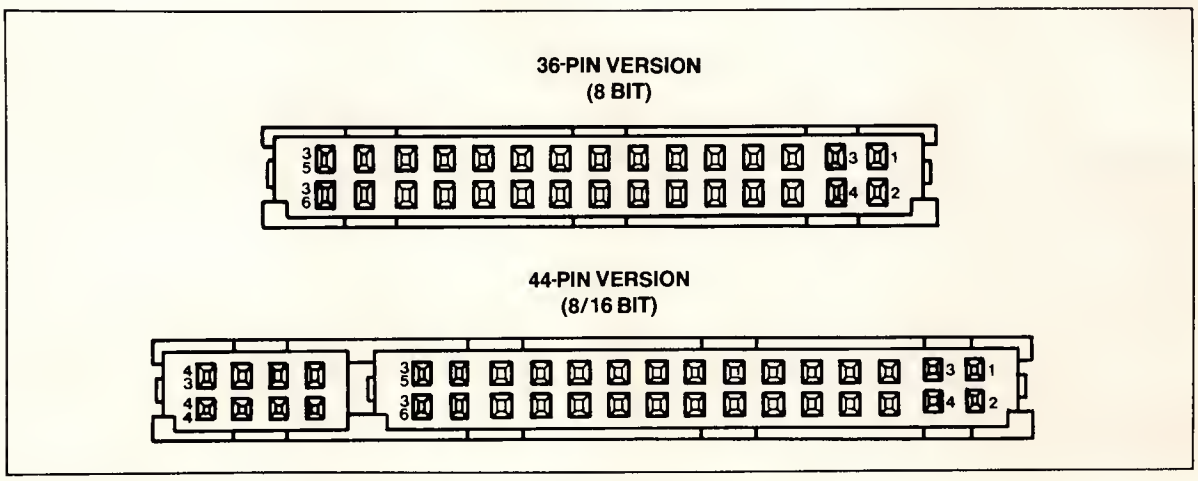


Figure 17. Connector pin numbering (top view of baseboard connector).

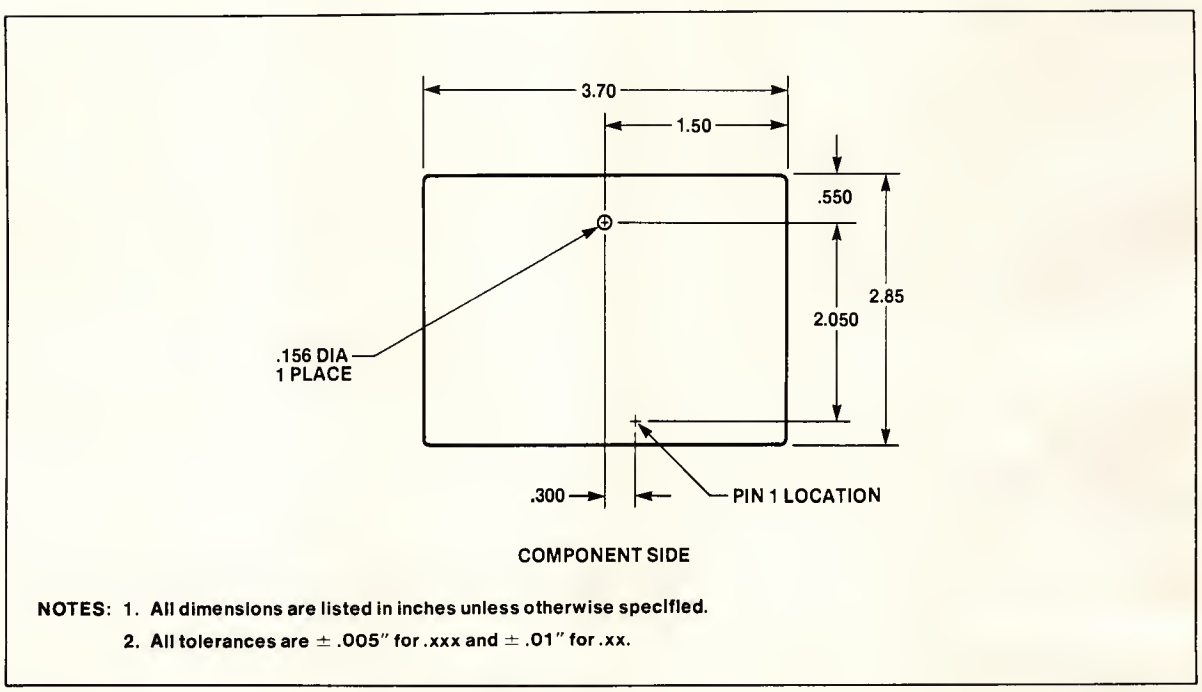


Figure 18. Standard outline for single-wide expansion module.

6.3.3 Interlocked operation

F represents a baseboard that does not support interlocked operation (that is, only full-speed operations are performed).

I represents an expansion module that requires interlocked operation (that is, requires the baseboard to support MWAIT*).

6.3.4 Examples

A 16-bit baseboard that supports both 8- and 16-bit expansion modules, that has DMA capability, and that supports the MWAIT* function would be specified as follows:

P959 I/O Expansion Bus Compliance: D16/16 DMA



PROPOSED STANDARD SUBJECT TO REVISION

Table 7.
Pin assignments.

Pin	Mnemonic	Description	Pin	Mnemonic	Description
1	+12V	+12 Volts	2	−12V	−12 Volts
3	GND	Signal Ground	4	+ 5V	+5 Volts
5	RESET	Reset	6	MCLK	Expansion Module Clock
7	MA2	Address 2	8	MPST*	Expansion Module Present
9	MA1	Address 1	10	—	Reserved
11	MA0	Address 0	12	MINTR1	Interrupt 1
13	IOWRT*	I/O Write Command	14	MINTR0	Interrupt 0
15	IORD*	I/O Read Command	16	MWAIT*	Expansion Module Wait
17	GND	Signal Ground	18	+ 5V	+5 Volts
19	MD7	Data Bit 7	20	MCS1*	Chip Select 1
21	MD6	Data Bit 6	22	MCS0*	Chip Select 0
23	MD5	Data Bit 5	24	—	Reserved
25	MD4	Data Bit 4	26	TDMA	Terminate DMA
27	MD3	Data Bit 3	28	OPT1	Option 1
29	MD2	Data Bit 2	30	OPT0	Option 0
31	MD1	Data Bit 1	32	MDACK*	DMA Acknowledge
33	MD0	Data Bit 0	34	MDRQT	DMA Request
35	GND	Signal Ground	36	+ 5V	+5 Volts
37[1]	MD14	Data Bit 14	38[1]	MD15	Data Bit 15
39[1]	MD12	Data Bit 12	40[1]	MD13	Data Bit 13
41[1]	MD10	Data Bit 10	42[1]	MD11	Data Bit 11
43[1]	MD8	Data Bit 8	44[1]	MD9	Data Bit 9

NOTE 1: MD8-MD15 used only on 16-bit systems.
NOTE 2: Signals ending with an asterisk (*) are active-low. Signals ending without an asterisk are active-high.



PROPOSED STANDARD
SUBJECT TO REVISION

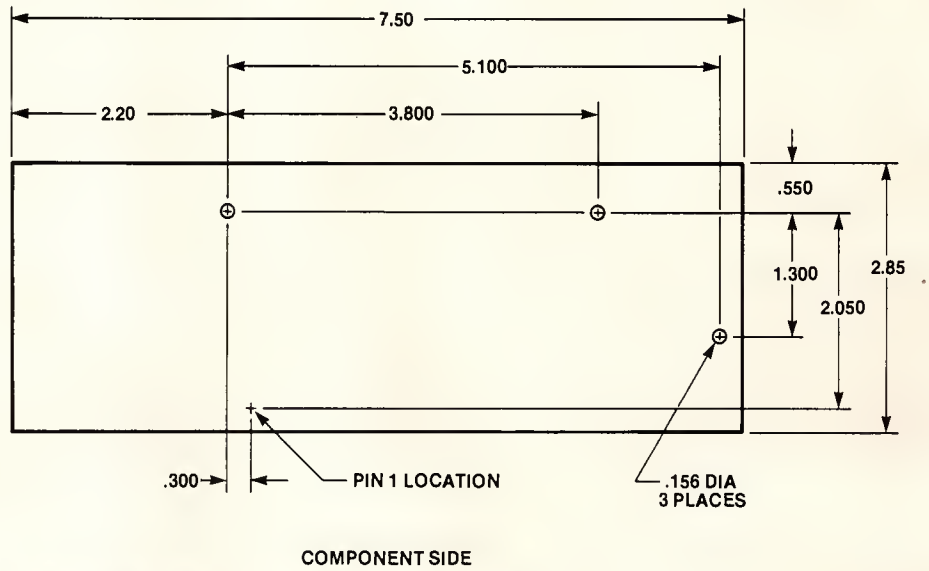
An 8-bit expansion module that does not require the MWAIT* line and does not support DMA would be specified as follows:

P959 I/O Expansion Bus Compliance: D8

6.3.5 Compliance marking

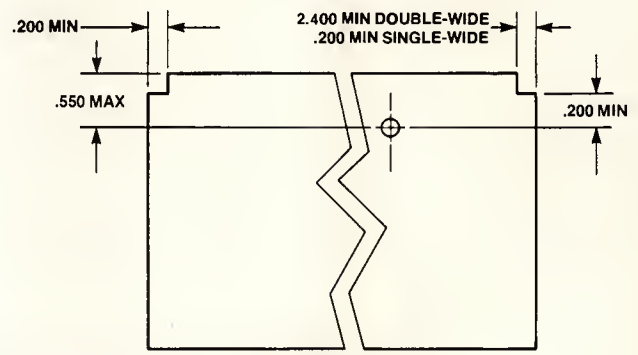
The compliance levels of a product shall be documented in all product specifications and optionally marked on the printed circuit board.

Appendices A and B are omitted to save space—see note following the Table of Contents.



NOTES: 1. All dimensions are listed in inches unless otherwise specified.
2. All tolerances are ± .005" for .xxx and ± .01" for .xx.

Figure 19. Standard outline for double-wide expansion module.



NOTES: 1. All dimensions are listed in inches unless otherwise specified.
2. All tolerances are ± .005" unless otherwise specified.

Figure 20. Edge-finger connector specifications.

The Object Pascal Architecture provides 22 simple stack instructions which enable straightforward compilation of Pascal-like languages.

A Reduced High-Level-Language Instruction Set

Peter U. Schulthess

Federal Institute of Technology
Zürich, Switzerland

Compiling high-level-language programs into low-level machine code is a complex task. However, it can be broken down into smaller pieces if the program is first compiled to a suitable intermediate instruction set. This intermediate code representation can then be either interpreted or further translated to final machine code.

Machine code can be generated for several different computers from a single intermediate representation. The Uncol machine[1] was the first proposal for an intermediate program representation that would facilitate language implementation on different computers. The Pascal p-code instruction set[2] was designed for a hypothetical stack computer to verify the code generation of a CDC Pascal compiler,[3] but then served as a tool for hundreds of compiler transportation projects. Such an instruction set is closely patterned after the source language to be compiled, and the intermediate code generation can be kept simple. The Pascal-P compiler is substantially smaller and less complex than a compiler for the CDC 6000 series or for the IBM System 370.[4]

Reducing the complexity of a compiler is worthwhile as soon as the cost of the language system approaches the cost of the hardware. In some cases a complex compiler may not even fit into the small computer and a cross-compilation on a different machine becomes necessary. As a consequence, it is tempting to build hardware that directly interprets such an instruction set if it can be done at a competitive cost. Dozens of high-level-language architectures have been proposed, simulated, and built in the last ten years.

Optimal instruction encoding. The generated code for a language-oriented instruction set is typically two to four times more compact than for a conventional register instruction set. This situation is frequently reported in the literature.[5-13]

Dense code is achieved by selecting short container sizes for frequent opcodes, addresses, and constants. Often a single language-specific instruction (LOOP, CASE, MARK STACK) performs the work of many pri-

0272-1732/84/0600-0055$01.00 © 1984 IEEE

mitive instructions. Dense code can lead to faster program execution because fewer instruction fetches are required.

The Burroughs B1700 computer series is dynamically microprogrammable and defines a number of different instructions sets (S machines).[14] Each one is optimized for a particular language or application environment. Very compact code is achieved for each language separately. Other language-oriented instruction sets which yield dense object code are described by Best et al.,[5] Tanenbaum,[12] Wilner,[14] Clark,[15] and Wirth.[16]

Glossary

The *architecture of a computer* is its structure as far as it is relevant to the compiler. It includes a description of stacks, registers, data formats, and instruction formats. On the other hand, it normally ignores aspects such as cache memories and instruction overlap.

P-code is the instruction set of a simple hypothetical stack computer that is particularly suitable as a target machine for Pascal. Typically, this stack computer is not realized as a piece of hardware but rather as an *interpreter* program that runs on another machine.

Dynamic microprogramming is a technique used to alter the microprogram and thereby the instruction set of a machine when switching from one program to another.

Lexical levels represent the static nesting of scopes in procedures and blocks as it can be determined by the compiler. There is a global lexical level and a current lexical level. The latter contains the variables, names, and temporaries of the currently executing block.

The *display* is a table of fixed size containing pointers to the start of all currently accessible levels.

Occurrence designates the position of a variable on its lexical level.

A *data segment* is collectively formed by the data items on a lexical level.

Code segmentation is a technique used to partition the object code of a program into separate segments in order to keep only the currently required code segments in main memory.

A *data structure* is implemented in Pascal as a record type or as an array type. The name of the record allows us to refer collectively to the fields within the record even if these are of different types (elements of an array are all of the same type).

Type descriptors reside in the type table and contain the size, type, dimensions, and offset of fields within a record. The entries in the type table are typically created by the compiler and remain unchanged during the execution of the program.

Data descriptors are created at procedure or block entry and during manipulation of data. They reside on the descriptor stack. Non-self-relative descriptors can point to a type descriptor in the type table.

The *heap* is a storage area where variables can be explicitly allocated and deallocated irrespective of the nesting of executing procedures. In Pascal, the data in the heap are accessed by pointers and generated by the standard procedure NEW.

Template descriptors are a special kind of non-self-relative data descriptor that requires a pointer value from the stack to complete the addressing process. Multiple instances of a given data type in the heap are all accessed through the same template descriptors.

Complex vs. reduced instruction set machines. Two particularly interesting computer architectures have recently been implemented as VLSI circuits. The AAMP—Advanced-Architecture Microprocessor—developed by Rockwell International[5] is a microprogrammed implementation of an elaborate high-level-language machine containing 153 different stack instructions. It is representative of a class of a complex instruction set machines oriented towards block-structured high-level languages. For a microcomputer, the machine also exhibits impressive floating-point performance.

In contrast, the RISC—Reduced Instruction Set Computer—developed at Berkeley[17] is not microprogrammed and has only 30 instructions, which operate on a set of registers. In the RISC, the chip area that in a complex instruction set processor is required to implement the microprogram is traded for a large pool of registers. The register pool is used in a clever addressing scheme called the *overlapping register window*. The scheme is slightly reminiscent of a single run-time stack and contributes significantly to the high execution speed of the RISC.

A heated discussion is currently taking place as to whether the cost of implementing a complex instruction set can be justified when many of the numerous and complex instructions will be rarely used. It is argued that additional instructions require little additional space on the chip and share microroutines with the basic instructions. It is also argued that the complex instruction set is justified by the additional program protection and simplified software it provides, and that these advantages outweigh any consideration of "raw" execution speed.

It is not clear how the RISC would be affected if the instruction set were to include floating-point arithmetic. (There would be enough free opcodes, at least.) The object code for the RISC is less dense than for a typical high-level-language machine.

We admire the astonishing results of modern computer architecture and in particular VLSI research. Designs like the NS16032, for instance, are noteworthy. Motivated by these results, we present a hypothetical OPA—Object Pascal Architecture—as a candidate for a reduced high-level-language instruction set computer. With such a machine, we hope to combine the advantages of complex instruction set computers (CISCs) and reduced instruction set computers (of which Berkeley's RISC is an example).

The OPA machine

This article describes the OPA concept and its current implementation as a software interpreter. The acronym OPA—for Object Pascal Architecture—was chosen because OPA is suitable as a target for object code generation from Pascal-like languages and also because it manipulates data objects rather than typeless values and machine addresses.

The OPA machine has been influenced by the B1700 SDL intermediate architecture in its use of multiple run-time stacks, data descriptors, and code segmentation.[14,18]

It is more general-purpose, however, and can well support most block-structured and strongly typed languages. (We aim for support of Ada in the end.) Simple solutions have been found to handle structured data. OPA uses a different set of run-time stacks than the SDL, and the architecture detects most errors connected with uninitialized variables and subrange violations. Instead of the approximately 140 instruction codes in the SDL architecture, the OPA machine needs only 22. It is, therefore, a high-level-language machine with a reduced instruction set.

We have written a compiler from Pascal to OPA code that consists of only 1900 Pascal statements and compiles itself to a total of 7000 instructions and 12.5K bytes of OPA code. Most Pascal compilers are between two to ten times longer in terms of statements and code size. The compiler is described in a separate paper.[19]

The advantages of the OPA are simple code generation, extensive run-time checks, and compact object code, particularly for large programs. These advantages are achieved by a somewhat unusual machine structure that separates data-descriptive information into static type descriptors and lexically allocated data descriptors.

Data descriptors. We start our description of the OPA with the formats of the data descriptors (Figure 1). The descriptors are 64 bits long and all carry a tag field. Tag values between 0 and 5 mark a self-relative descriptor for a scalar data item (CHARACTER, INTEGER, REAL, SET, . . .), where the value is stored in the descriptor itself. (We adopt the Burroughs terminology of *self-relative* and *non-self-relative* descriptors rather than using the words immediate and indirect.) Tag values 5 and 6 are reserved for a later extension of the OPA concept to include varying-length strings (see again Figure 1).

A descriptor with a tag value of 7 describes a data item that is either an indirectly accessible scalar or a data structure. Such a non-self-relative descriptor will not hold the value of a variable but only its description. The fields in a non-self-relative descriptor (tag = 7) carry additional information to describe the data structure:

- The *data address* points to a location where the actual value can be found. The current interpreter uses only 16 of the 32 address bits. The address designates a byte location.
- The *data type index* specifies an entry in a separate type table where the structure information of the record and the array dimensions can be found. (Addressing through the type table will be described later in this article, under "Type Tables.")
- An *initialization link* permits an initialization test for scalar variables that are passed by reference. The link points to the original descriptor and its initialization status. (See "Initialization Checks.")

The descriptor size has been set at 64 bits because this is about the minimum word size that allows for a decent implementation of the SET OF CHARACTER type. It leaves an adequate precision margin for arithmetic operations and a large address range in the non-self-relative descriptor.

Run-time stacks. The instructions of an executing OPA program operate on a set of functionally separate run-time stacks. Figure 2 shows the stacks and tables that are relevant to the accessing of program data. The run-time stacks are listed in Appendix B.

Descriptor stack. The descriptor stack contains a data descriptor for each declared variable or type identifier. These descriptors are grouped into lexical levels and directly stacked on top of each other without an intervening stack mark. The top of the descriptor stack is also used during the evaluation of expressions to pass parameters to procedures and to return the result of a function procedure. There is no separate expression stack.

Data stack. The data stack contains the values of structured types like arrays, records, and strings. The internal register pointing to the top of the data stack is incremented for each lexical allocation of a structured type. The previous value of the stack pointer is recorded in the allocated descriptor as the data address. A suitable piece of the data stack is also allocated if a long temporary value is loaded onto the descriptor stack during expression evaluation. While the descriptor stack contains 64-bit words, the storage allocation in the data stack is currently in 8-bit portions. A section of the data stack is used as a heap. The management of the heap is left to the language implementer.

Display. Each display entry can point to a currently accessible lexical level in the descriptor stack. Strictly speaking, the display is not a run-time stack but a table whose size can be determined by the compiler. Our implementation uses a display table with 16 entries.

A static chain to link the lexical levels could have been used as an alternative to a display table. We chose a display because it not only offers a means to implement the addressing scheme of nested procedures but, in the future, will also support the addressing of data segments in external modules. This is important because modern languages tend to bundle their variables in modules and packages rather than in nested procedures.

Block stack. Recursive activation of a procedure creates multiple generations of the same lexical level. Successive allocations will overwrite the previous entry in the display table. The previous display entries are saved in the block stack to permit later reactivation of recursively buried procedures. The lexical-level number, the current position of the data stack pointer, and a pointer to the top of the return stack are also saved to help the GOTO OUT OF BLOCK. (See "Control Transfers.")

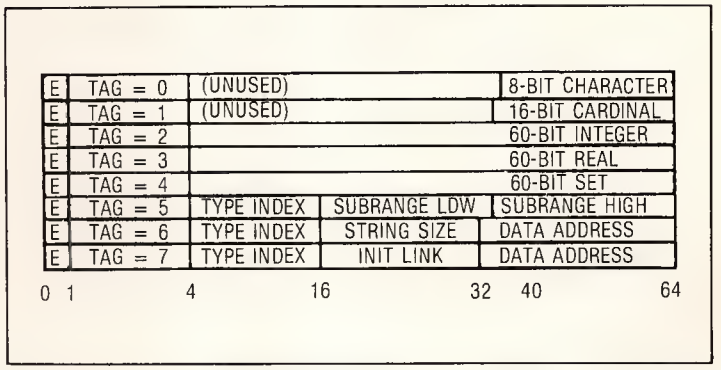


Figure 1. Format of the data descriptors (E = empty bit).

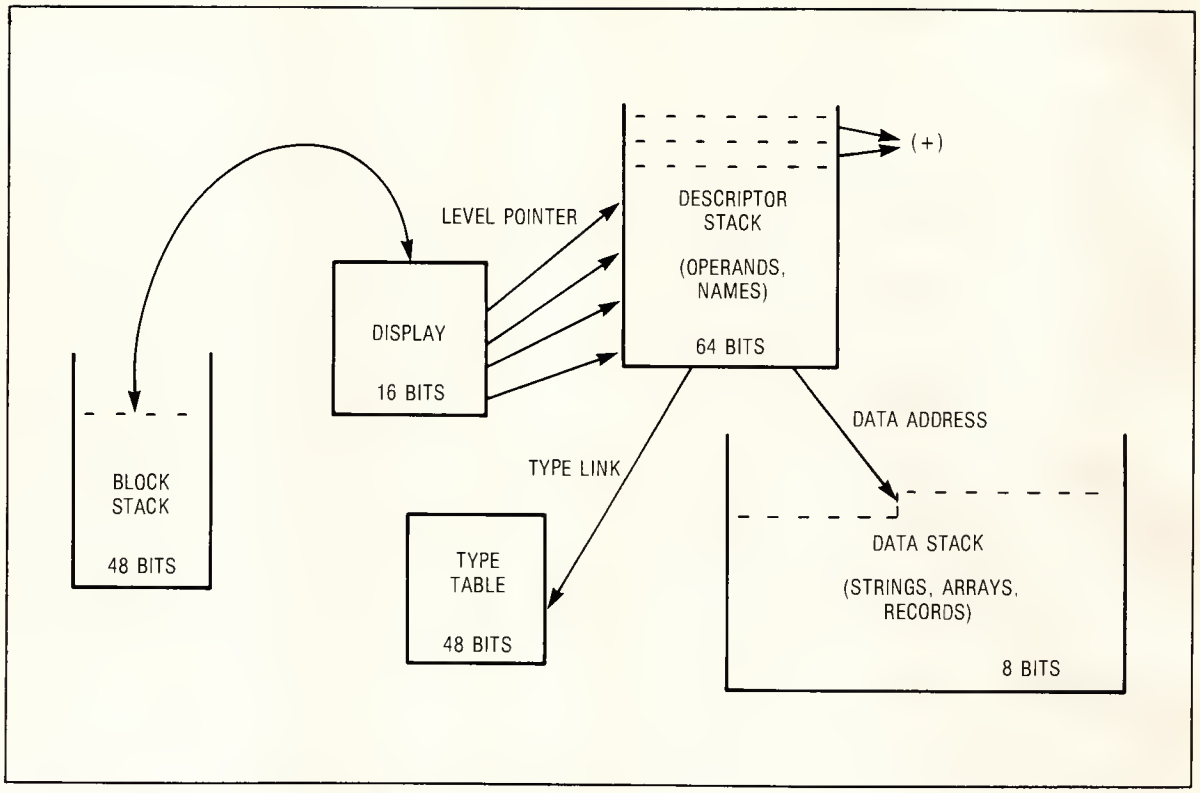


Figure 2. Structures involved in the data accessing mechanism.

Return stack. The return stack is not shown in Figure 2 since it does not participate in the data accessing mechanism but contains code addresses only. These addresses are pushed on the return stack by a previous procedure call, case selection, or program loop. Since the object code is structured into code segments, the code addresses on the return stack consist of an 8-bit code segment number and a 16-bit displacement within the segment.

Addressing of data. The data address in an instruction consists of three 4-bit portions and follows the 4-bit opcode (Figure 3). Instructions will be discussed in the next section.

Lexical levels. In the data address portion, the *field number* is used only when a field in a record or an array element must be selected (see "Field Number"). The lexical-level field selects an entry in the display table and therefore the starting point of a group of descriptors. The occurrence number then selects a single data descriptor, and, in the case of a LOAD VALUE instruction, a copy is produced on the stack.

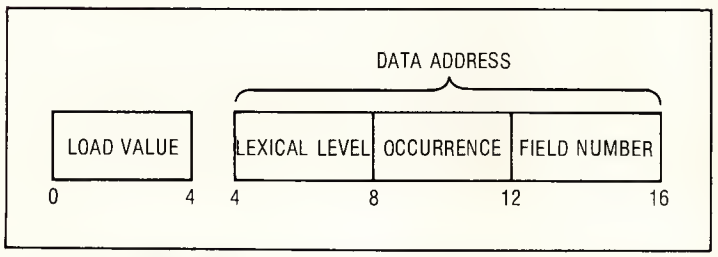


Figure 3. Data address in a LOAD VALUE instruction.

The 4-bit fields provide for 15 lexical levels and for 16 occurrence descriptors on each level. Successive display entries are used if the source program declares more than 16 variables on one level. Typically, three to seven display entries will be used for the global level. The intention here is to keep a simple and uniform address format and to avoid a distinction between LOAD GLOBAL, LOAD INTERMEDIATE, and LOAD LOCAL. Our experience shows that 15 levels of 16 descriptors are plenty. (For pathological cases an escape to 256 levels is possible, as indicated in Appendix A.)

Pointers. Non-self-relative descriptors describe either structured variables like arrays and records or variables that may be scalar but have been passed as reference parameters.

An obvious choice to implement a pointer addressing scheme would be to use non-self-relative descriptors as pointers to dynamically allocated structures in a heap. Unfortunately, this would require additional instructions to manipulate descriptors directly and thus would defeat the purpose of the reduced instruction set. Separate instructions would be required to load the value of a pointer and to load the record being pointed at.

Our method of pointer addressing is based on the use of special *template descriptors*. These are non-self-relative data descriptors with a value of zero in the data address field. One template descriptor is allocated for each declared or implied structured type, but no space in the data stack is initially allocated.

An instruction referencing a template descriptor will notice the zero address field and automatically fetch a

pointer from the top of the stack. This pointer will normally resemble a self-relative integer loaded by a previous LOAD VALUE instruction.

The type index field in the descriptor indicates the allocated record type and prefixes the actual allocation in the heap. This allows for a loose test at run time to determine that the pointer actually points to a correct allocation. This test could be made tighter at the expense of additional heap storage.

Type table. Non-self-relative descriptors include a type index field that points to a type descriptor in the type table. Figure 4 gives the format of the entries in this table.

The range entry either describes an array dimension by its low bound, high bound, and type of array element, or it describes the valid range for a subrange type (e.g., AGE = −1 . . . 110).

The field entry specifies the size of a record field and an eventual *field offset* within the record structure. The *field type* indicates the arithmetic type of the field (tag0 . . . tag4) or, if the field is a record structure itself, points to another entry in the type table.

Groups of type descriptors are linked in a tree-like fashion, with all fields in a parent record directly linked to the immediate parent. The same link field is also used to connect the successive range entries of multidimensional arrays. The top entry for an array is always a field entry specifying the total size of the array and an offset of zero. The link field can link to a type entry up to 127 locations above. Its first bit indicates whether the entry is a range or a field (R = 1/F = 0).

Variables of the same type share the entries in the type table and reduce the required memory space. Our current implementation uses a type table that is prepared by the compiler and that remains constant during the execution of the program. If fully dynamic types are required, the type descriptors must be subjected to the same lexical mechanism as the descriptors on the descriptor stack and the data stack.

Field number. Creating a descriptor on the descriptor stack for each field in a record and for each array dimension would be too expensive, take too much execution time, and soon exhaust the 16 available entries on each lexical level. A data descriptor is allocated only for an entire record, and all its fields are found via the field entries in the type table (Figure 5). The algorithm of this walk through the type table is given by the routine DESCRIBE in Appendix C.

In Figure 5, the field A.f3 is accessed by a LOAD VALUE instruction. The identifier A.f3 designates the third field in the record variable A. The data descriptor for A points to entry 23 in the type table. To this value of 23, the field number from the instruction is added, giving a total of 26.

To address the field A.f3, the machine walks through the type table from entry 26 to entry 23, computing the address at each step. In our case, only one step is required because the field entry for f3 is directly linked to A (like all fields on its record level). In the case of nested record declarations and multidimensional arrays, several steps become necessary. When the machine steps through array dimensions, an index is automatically popped from the stack and checked against the specified limits.

Normally, up to 14 distinct field numbers per record nesting level are possible, but an escape mechanism permits field numbers up to 255 (see Appendix A). The format of our type table, together with the field number mechanism, can economically represent and access complicated data structures and multidimensional arrays without a complex code generation pattern in the compiler. The scheme can be augmented at the expense of some extra storage to check for a correct tag field when accessing fields in record variants.

Instructions. The OPA architecture defines 22 instructions that operate in a straightforward manner on the runtime stacks. The formats of these instructions are given in Appendix A. We can distinguish four categories of instructions—those for data manipulation, those for data movement, those for data allocation, and those for control transfers.

Data manipulation. There are six instructions for data manipulation:

ADD
SUBTRACT
MULTIPLY
DIVIDE
MODULO
COMPARE

These instructions pop two operands from the top of the descriptor stack and push the result back on. For long operands that cannot reside in a self-relative descriptor, the data stack is also involved.

OPA instructions are generic in the sense that the actual operation is finally determined by the type of operands. Depending on the operand tag fields, an ADD instruction will produce a floating-point sum, an integer sum, or the union of two sets. If an integer is added to a set, then the ADD instruction will perform the logical include operation. We have not implemented all combinations of tag fields, but, in principle, a generic ADD instruction far exceeding the scope of the Pascal language could be defined. One might suggest that the ADD instruction would concatenate two character strings if the appropriate tag fields were encountered. The COMPARE instruction carries a 4-bit compare code specifying a test for equal, unequal, less than, and the like. The other instructions are self-explanatory.

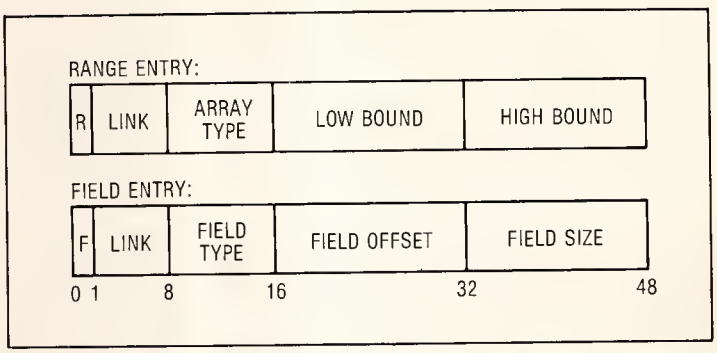


Figure 4. Type table formats for array dimensions and record fields.

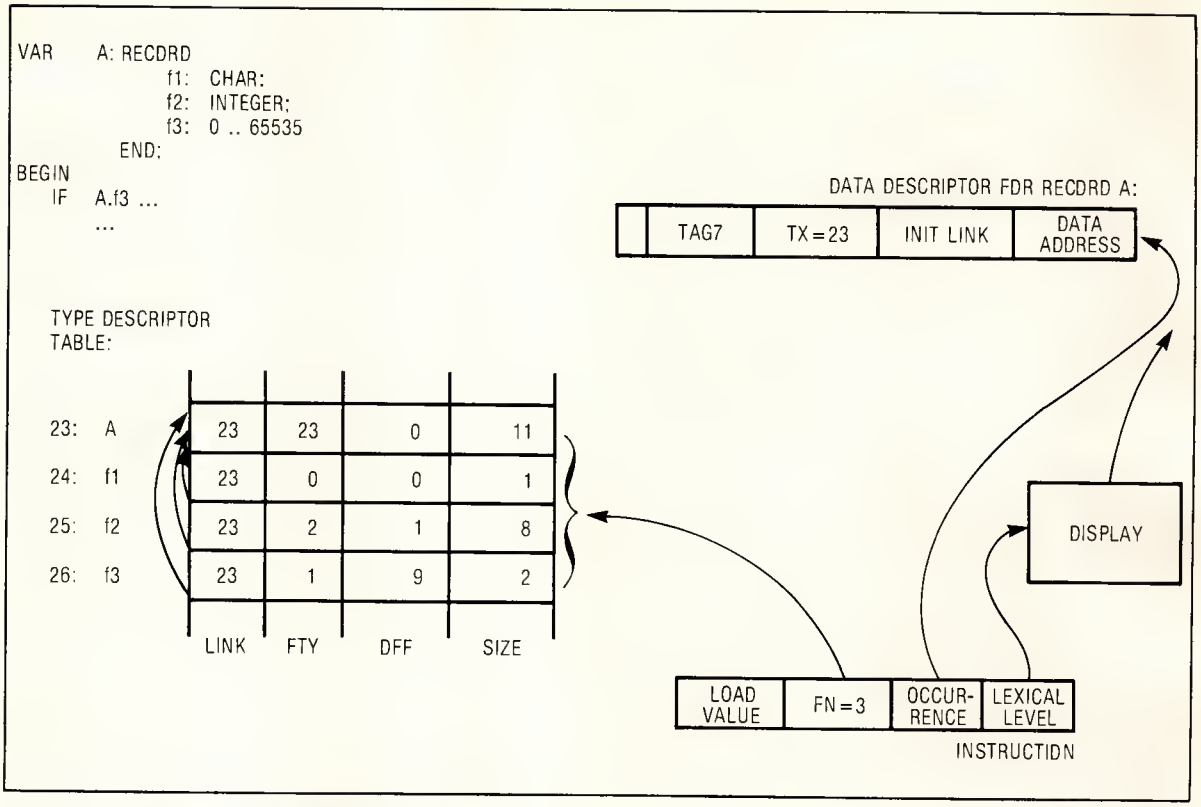


Figure 5. An example of the use of the field number.

Data movement. There are five instructions for data movement:

LOAD SHORT CONSTANT
LOAD CONSTANT
LOAD VALUE
LOAD ADDRESS
STORE

These instructions load or store data to or from the top of the stack, respectively. Characters and short constants with a value between zero and 95 are loaded by a one-byte instruction (type = 0). The type of other constants is indicated by a type index in the LOAD CONSTANT instruction, and information such as length is fetched from the type table. The actual value of the constant follows in line. (Loading a structured constant or one of its elements could be implemented in a natural way.)

LOAD VALUE loads a copy of a variable on the top of the descriptor stack and, if necessary, on the data stack. For scalar operands, a self-relative descriptor is always produced on the stack. The opcode of the instruction is followed by a 12-bit data address (see again Figure 3). If the field number is not zero, the type table is examined and the necessary indices and pointer values are popped from the stack.

LOAD ADDRESS always loads a non-self-relative descriptor pointing to the original data or some element of it. The init link field in the new data descriptor points to the original descriptor. LOAD ADDRESS is mostly used to pass reference parameters to procedures and WITH clauses.

The STORE instruction stores the value on the stack into the specified data address. Indexing and pointer qualification on the target location are performed if necessary. A conversion takes place if, for instance, an integer value is stored into a real variable. There is currently only a 16-bit format for the LOAD VALUE, the LOAD ADDRESS, and the STORE instructions.

Data allocation. There are four instructions for data allocation:

ALLOCATE VARIABLE
ALLOCATE TYPE
MARK STACK
UNMARK STACK

These instructions are concerned with allocation and deallocation of descriptors. The operation code for ALLOCATE VARIABLE is followed by a type index. It constructs a new data descriptor on the stack according to the information it finds in the type table. For records or arrays, a portion of the data stack is also allocated. The empty bit of the new descriptor is set TRUE.

ALLOCATE TYPE creates a descriptor on the stack for a structured data item. No space is reserved in the data stack, and the address field in the descriptor is set to a special value of zero. This special data address will indicate to a later addressing cycle that a pointer is needed from the stack (see again section headed "Pointers"). Both kinds of allocate instructions can, of course, share the same type table entries.

The MARK STACK operation should also be discussed under the heading of data allocation. In the OPA architecture, the stack position must be remembered for the

descriptor stack and for the data stack. The stack is always marked after the parameters have been loaded and after the call. The MARK STACK instruction occurs in the body of the called procedure and not at each code location from which the procedure is called. This can save many instructions. MARK STACK not only preserves previous stack pointers but also does a range check and eventually converts the procedure parameters to the required type.

If the stack is marked for a function data segment, then the value to be returned on exit from the function is allocated at the bottom of the new data segment, and it is addressable from inside the procedure as occurrence zero. The stack pointers saved by the MARK STACK instruction are adjusted in such a way that a later UNMARK STACK instruction will leave the function value on the stack. The same UNMARK STACK instruction either ends a regular procedure or leaves the function value on the stack.

Control transfers. There are seven instructions for control transfers:

CALL
LEAVE
LEAVE CONDITIONAL
SELECT CASE
FORMAL CALL
LOOP
GOTO

These instructions operate primarily on the return stack, which contains return addresses from procedure calls, loops, and case variants. Instructions do not directly carry code addresses but only indices into the label table (see "Label Segments," below). This simplifies code generation in the compiler.

The CALL instruction pushes the next instruction address onto the return stack and then branches to the location to be called. Lexical-level information and the other stacks remain unchanged. FORMAL CALL pops an integer value from the stack and interprets it as a segment number and a label index. This code location is then called.

SELECT CASE pops an argument from the descriptor stack. If the argument is within the given range, then an element is selected from a list of indices which follow in line after the instruction. If the argument is not in the range, then the ELSE INDEX is selected. A return address is pushed onto the stack, and the selected case clause is called just as it would be by a CALL instruction. If the value of a selected list index is 255, then the SELECT CASE instruction is skipped (="no call"). The SELECT CASE instruction also implements IF-THEN-ELSE statements.

The LOOP instruction jumps to a given code address without touching on lexical levels or the return stack. GOTO can go out of loops and blocks and adjusts the runtime stacks appropriately. Restoring the display after a GOTO OUT OF BLOCK is a nontrivial task; it amounts to a repeated UNMARK STACK operation until the desired lexical level is found on the block stack. The return stack and the program counter are then adjusted with the help of a special field in the block stack. With a static chain instead of a display table, the GOTO OUT OF BLOCK would have been simpler.

The LEAVE instruction pops a return address and uses it as such. It might leave a program loop or an ELSE clause. No adjustment of the lexical level is done.

LEAVE CONDITIONAL pops an argument from the descriptor stack and performs a LEAVE operation if the popped argument was TRUE.

Object code structure

Code segments. A segment contains the code of one or more procedures or modules. Program code is organized into several logical segments. A virtual memory mechanism is implemented so that only the segments belonging to the working set must actually remain in memory. The presence of a code segment needs to be checked only by the control transfer instructions, which amount to 24 percent of all instructions. The associated overhead for the code address translation in the interpreter is small.

Label segments. Each code segment has its own label segment with up to 255 label entries. Each entry consists of a code segment number and a displacement within the segment (Figure 6).

The control transfer instructions do not have in-line code addresses but rather indices into the label table for the current segment. These indices are usually 8 bits long and create the restriction of 255 label entries per segment. The 8-bit label indices are convenient because they result in short selection lists in the SELECT CASE instruction. Furthermore, the code generator always knows the size of the branch addresses in the instruction format, even for forward-referenced locations. No sophisticated address fix-up is required.

It is possible—but rare—that the same target code location will be referenced by several label entries in different segments, creating redundant label table entries. On the other hand, several instructions will often make use of the same entry in the label table. This is an instance of information sharing analogous to the situation in which several data descriptors share the same type table entries. In our compiler, one label entry is referenced by 2.6 instructions.

Suboptimal encoding. Most language-oriented instruction sets achieve dense code through a frequency-dependent encoding of instructions and instruction groups. In particular, the frequency of small constants and short addresses is considered.

To avoid a proliferation of opcodes, we chose a suboptimal instruction encoding (see again Appendix A). But even with such an encoding, we can nevertheless achieve good code density through an appropriate choice of runtime organization and instruction semantics. Some Pascal program fragments and the resulting code sizes are shown in Table 1. The corresponding instruction sequences are given in Appendix D.

The examples in Table 1 and in Appendix D clearly show that our choice of a single 12-bit data address format consisting of field number, occurrence number, and lexical level is suboptimal. The LOAD VALUE, the LOAD ADDRESS, and the STORE instructions are 16 bits long even when a local variable can be accessed with a short offset. However, even with our 12-bit data address, we

obtain competitive code density (see examples discussed in "Code Density," below).

Since our 12-bit address also contains an eventual field number, we can only use a short address format when the variable is addressed through field number zero and when it is on the local level. We were surprised to find that fewer than 25 percent of all data addresses in our programs were candidates for a short address format.

To anyone who needs maximum code density at the expense of a *more complicated instruction set*, we suggest two alternatives:

- The LOAD SHORT CONSTANT instructions from 48 to 95 can be replaced by short variants of LOAD VALUE, LOAD ADDRESS, and STORE, each carrying only a 4-bit occurrence number into the current lexical level. The short encoding of 48 to 95 does not pay off anyway.
- A context-dependent encoding of the data address can be introduced in spite of the resulting problems for the debugger. The interpreter will first consider the lexical level. If the indicated level is greater than the currently active one, the value is considered as an occurrence indication on the current level (occurrence $\leq$ lexiclevel $-$ currlev). A further context dependence can be introduced by providing the field number only if a non-self-relative descriptor is accessed ($\sim$40 percent of the data addresses). This is possible because the compiler can always determine whether a descriptor will be self- or non-self-relative.

We do not recommend either of the above alternatives, however, but prefer our original scheme because it is easier to understand and to implement.

The last statement of Table 1 is an example in which the OPA machine allows for particularly elegant and compact code. The detailed code pattern for the CASE statement is given in Appendix D. We capitalize on the fact that in the OPA machine a case selection is consistently patterned after the procedure call. A short case list is achieved, and the procedure is called in one step instead of two. This also saves return stack space. Incidentally, the same benefits and code generation patterns arise in the IF-THEN-ELSE constructions.

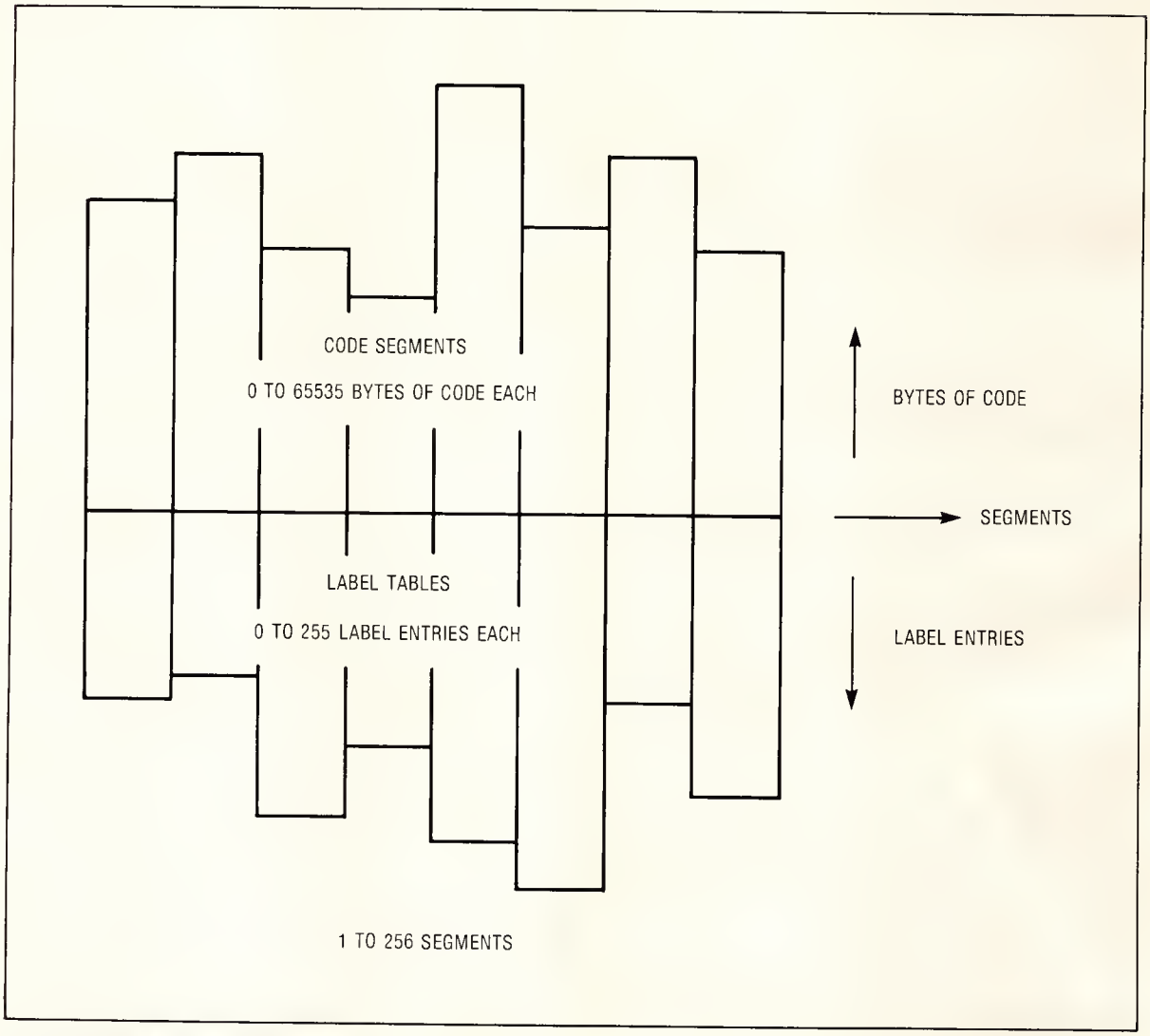


Figure 6. Code and label segments.

Run-time checks

Array bound checks. Array bound checks are performed during the type table walkthrough and do not require explicit check instructions in the code. Whenever a subscript is popped from the stack, it is checked against the low and high bounds in the type table entry. If the low bound is not zero, then its value is subtracted from the subscript. The subscripts of a multidimensional array are checked individually. If the size of an array element is not one, the subscript is multiplied by the size of the array element.

Range checks. The range entries in the type table normally specify an array dimension. They can, however, represent the permissible range of a subrange variable or of an enumerated type. A non-self-relative descriptor pointing to this range entry will then be allocated for the variable instead of a self-relative one. This method of range checking will eventually be implemented by a compiler option and will require additional functionality in the interpreter but no further instructions. Currently, the self-relative descriptors only limit (and trap) their values to 0...255, to 0...65535, or to a signed 60-bit value.

Initialization checks. The initialization status of a variable is indicated by the first bit in the tag field—the empty bit. This bit is set to TRUE when the descriptor is allocated but not yet given a value. A later assignment will reset the empty bit. The empty bit mechanism catches all attempts to load an uninitialized scalar. For an array or a record structure, there is only a single empty bit. It indicates that none of the components have yet been assigned a value. A value of FALSE leaves open the possibility that some elements have remained uninitialized. Since one single descriptor is allocated for an array or a record variable, we can only check if at least one of the array elements or record fields has a defined value. This makes for an incomplete initialization test in the case of arrays and records.

The self-relative data descriptors deal only with the empty bit. Non-self-relative descriptors also contain an initlink field that points either to their own descriptors or to previous descriptors. This permits a similar initialization test even for indirectly accessed data. Instead of testing the empty bit of the descriptor being accessed, the interpreter tests the empty bit of the descriptor pointed at by the initlink. An initlink to a previous descriptor will normally be set up when a LOAD ADDRESS instruction loads a reference to a previous allocation or a field thereof.

Exceptions. If a run-time error is detected, label 0 of the current segment is called with the type of the error as a parameter. This label entry then calls an error handler that may write a memory dump to disk, go into an interactive debugging mode, or simply ignore the exception. The action of the error handler in the run-time system is beyond the scope of the OPA machine. Asynchronous events from peripheral devices can be similarly treated as accidental procedure calls.

Evaluation

Code density. To obtain a measure of OPA code density, we have compiled the widely circulated puzzle program by Forest Baskett (which is reproduced in Jacobi[7]) and about half of the OPA compiler (scanner and initialization) on several machines. The results are given in Table 2. The values include storage for string templates, label tables, and type descriptors. The puzzle program is given in two versions: version A is the original, and in version B the triple FOR loops in the initialization sequence have been turned into a functionally equivalent procedure.

In the OPA, 40 percent of the table space is for type table entries and 60 percent for the label tables in each code segment. Carrying the descriptors along is justified because of the increased flexibility in the data formats. The fact that we use a rather crude instruction encoding does not prevent us from achieving dense code. Both the UCSD machine and the M code machine manipulate 16-bit units. Set operations that the M code was unable to handle were commented out in the last program. This has further reduced the M code size. Including the check instructions would add about 17 percent to the IBM 370

Table 1.
Object code for Pascal fragments (no checks, short local addressing).

	OPA CODE*	UCSD PASCAL[15]	M CODE[16]	IBM 370[4]
ch:= 0;	= 3 BYTES	= 3 BYTES	= 2 BYTES	= 6 BYTES
ch:= ' ';	= 3 BYTES	= 3 BYTES	= 3 BYTES	= 8 BYTES
A.f3:= i;	= 4 BYTES	= 5 BYTES	= 3 BYTES	= 8 BYTES
proc2(i,3);	= 4 BYTES	= 4 BYTES	= 3 BYTES	= 20 BYTES
IF ch<' 'THEN proc1;	= 7 BYTES	= 7 BYTES	= 7 BYTES	= 16 BYTES
CASE i OF 0,2,4,6: proc1; 1,3,5,7: proc3 END;	= 10 BYTES	= 31 BYTES	= 23 BYTES	= 64 BYTES

Table 2.
Compilation of complete programs (check instructions omitted).

	OPA CODE*	UCSD PASCAL[15]	M CODE[16]	IBM 370[4]
PUZZLE-PROGRAM A				
CODE BYTES	1240	2498	1310	~4804
TABLES	284	50	92	~164
TOTAL	1524	2550	1402	4968
PUZZLE-PROGRAM B				
CODE BYTES	731	1145	755	~2390
TABLES	189	60	94	~168
TOTAL	920	1205	849	2584
OPA COMPILER (~50%)				
CODE BYTES	6217	7903	6142	
TABLES	1296	1030	680	
TOTAL	7513	8930	6822	17256

code, about 10 percent to the UCSD Pascal p-code, about 5 percent to the M code, and zero percent to the OPA code.

Branching is efficient in terms of code density when a code location is called from several places. The entry in the label table can then be shared. Otherwise, the additional overhead for the 8-bit label index must be paid. Significant economies in code space occur in CASE statements that have long case lists with many elements referring to the same label.

The OPA architecture favors a programming style with many procedures that directly manipulate their parameters rather than local auxiliary variables. Access to local variables is not as profitable as in other machines. No short instruction formats that would warrant the allocation of these locals are currently available.

Code segmentation also contributes to an efficient utilization of the code space because only the working-set segments must remain in memory. This aspect is not represented in Table 2. The OPA compiler would discard its initialization segment of 2.5K bytes.

Data packing. Arguments favoring high code density are somewhat weakened by the decreasing cost of primary and secondary memory. Often, more memory will be required for storing data than for storing code. An instruction set should therefore economically handle and store a large variety of data formats and not just "words."

A bit-addressable instead of a byte-addressable OPA machine would use the same descriptor mechanism and the same instructions. Whereas direct addressing to the bit level is complicated on a 16-bit machine, it would present no problems with a 32-bit address in the non-self-relative descriptor. (The 64-bit descriptor could be squeezed to accommodate even a 48-bit address.) Memory access at the bit level is attractive for raster-scan displays. Current personal computers with raster-scan displays use either microprogrammed bit map instructions or a dedicated display processor.

The space taken up by non-self-relative descriptors in the descriptor stack is typically less than 6 percent of the whole data area, and these descriptors usually share many common entries in the type table. About 80 percent of the data area is spent on the data stack for allocating records and arrays (but not their descriptors). The space for descriptors in many cases is compensated for by a better memory allocation scheme for large data structures.

However, we worry more about the memory management problem created by the multiple run-time stacks than we do about extra space required for data and type descriptors. This memory management problem will eventually go away in machines with virtual memory systems. For the moment, we keep all stacks except the descriptor and the data stack movable. However, as soon as there are multiple processes in a machine, even one with a virtual memory system, the memory management problem will reappear.

Execution speed. We currently have an interpreter for OPA code written in Modula-2. Our purpose in developing this interpreter was to provide a tool for verifying the function of the OPA architecture. It runs on a 6-MHz MC68000 and on the Lilith Modula machine[16] and has an average instruction execution time of a little less than 500 microseconds on both machines.

This is clearly an unacceptable execution speed, and we should write an interpreter in assembly language so that the various registers and stack pointers of the virtual machine can be kept in real registers and not in memory. On a 10-MHz MC68000, we expect an average instruction execution time of around 40 microseconds. When emulated on a 16-bit machine, the 64-bit-wide descriptor stack requires many more memory accesses than an intrinsic 16-bit architecture. The number of memory accesses to the type table, however, remains relatively small because only about 10 percent of all instructions access the type table. The reason for the long execution time is primarily the large stack word and not the descriptor indirection. The time might be improved in a future version of the interpreter that would not always write all 64 bits.

In a hardware OPA machine, the 64-bit stack word might be squeezed down to a 32-bit word and the topmost stack locations could be kept in processor registers. A sophisticated OPA machine might overlap the execution of several instructions. Here, the separate run-time stacks and a read-only type table might be of some help—an instruction stepping through the type table would not fear a "last-minute" modification of the type entries. Instructions like CALL, LEAVE, and LOOP would handle their return addresses on the separate return stack and would not need to watch for an eventual push or pop by a previous incomplete data instruction, as might occur in a machine with a single stack.

Simplicity of the instruction set. We have substantially simplified the instruction set of a high-level-language machine and its encoding and have still achieved good code density. By introducing additional instructions, we could achieve even denser code, but we prefer to keep the instruction set in its simple form.

With the OPA, we hope to create an abstract view of a machine that is close to the programming language. One should be able to understand the semantics of the instructions without a detailed knowledge of what is happening in the micromachine. Since the functions are separated out into different stacks, they can be contemplated separately.

The reduction of the instruction set has shifted some of the burden from instruction processing to descriptor processing. This complexity appears in the interpreter routine DESCRIBE (see again Appendix D), which steps through the type table and also does the range checking. This routine consists of 18 statements. Additional complexity is also introduced in the arithmetic instructions, which must take into account the type of their operands. In our interpreter, this means a tedious extraction of 4-bit fields. In a hardware implementation, control signals for the ALU would not be derived from the opcode as usual, but rather from the tag fields of the operands.

The current OPA interpreter is 3100 bytes long on a Lilith personal computer.[16] Our next step will be to assess the complexity of an eventual hardware realization and the achievable speed. The small Pascal compiler we have written to generate OPA code indicates that code generation for the OPA is indeed simple.

High-level-language mechanisms can be implemented with a few simple instructions. We avoid the proliferation of opcodes that is traditionally introduced to achieve good code density. In addition, our high-level-language machine performs extensive run-time checks on the executing program and still allows for very compact object code. From a subjective point of view, we feel that the Object Pascal Architecture and its formats are clean and concise. The functionality achievable within the OPA extends far beyond the semantics of the Pascal language. We have implemented Pascal mainly for reasons of compiler simplicity.

The present implementation also proves that stack-oriented instruction sets can give very compact object code if a complete program—and not just the evaluation of arithmetic expressions—is considered. Arguments for and against the optimality of stack instruction sets are abundant in the literature,[8,10-12,20] but these arguments are not always based on sufficiently large and realistic program samples. ■

References

1. J. Strong et al., "The Problem of Programming Communication with Changing Machines," *Comm. ACM*, Vol. 1, No. 7, July 1958, pp. 12-18.
2. K. V. Nori, U. Ammann, K. Jensen, H. H. Naegeli, and C. Jacobi, "The Pascal-P Implementation Notes," in *Pascal—The Language and Its Implementation*, D. W. Barron, ed., John Wiley & Sons, New York, 1980.
3. U. Ammann, "Die Entwicklung eines Pascal Compilers nach der Methode des Strukturierten Programmierens," dissertation, ETH No. 5456, Swiss Federal Institute of Technology, Zürich, 1975.
4. G. Cox and J. Tobias, "AAEC Pascal Compiler for IBM 370 Machines," source text, Australian Atomic Energy Commission, Sydney, 1979.
5. D. W. Best, C. E. Kress, N. M. Mykris, J. D. Russell, and W. J. Smith, "An Advanced-Architecture CMOS/SOS Microprocessor," *IEEE Micro*, Vol. 2, No. 3, Aug. 1982, pp. 10-26.
6. H. G. Cragon, "An Evaluation of Code Space Requirements and Performance of Various Architectures," *Computer Architecture News* (ACM Sigarch newsletter), Vol. 7, No. 5, Feb. 1979, pp. 5-21.
7. C. Jacobi, "Code Generation and the Lilith Architecture," dissertation, ETH No. 7195, Swiss Federal Institute of Technology, Zürich, 1982.
8. J. Keedy, "On the Use of Stacks in the Evaluation of Expressions," *Computer Architecture News* (ACM Sigarch newsletter), Vol. 6, No. 6, Feb. 1978, pp. 22-28.
9. G. J. Myers, *Advances in Computer Architecture*, John Wiley & Sons, New York, 1978.
10. G. J. Myers, "The Case Against Stack Oriented Instruction Sets," *Computer Architecture News* (ACM Sigarch newsletter), Vol. 6, No. 3, Aug. 1977, pp. 7-18.
11. R. E. Sweet and J. G. Sandman, Jr., "Empirical Analysis of the Mesa Instruction Set," *Proc. ACM Symp. on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, CA, Mar. 1982, pp. 158-166.
12. A. S. Tanenbaum, "Implications of Structured Programming for Machine Architecture," *Comm. ACM*, Vol. 21, No. 3, Mar. 1978, pp. 237-246.
13. W. Wilner, "B1700 Memory Utilization," *AFIPS Conf. Proc.*, Vol. 41, 1972 FJCC, pp. 579-586.
14. W. Wilner, "Design of the B1700," *AFIPS Conf. Proc.*, Vol. 41, 1972 FJCC, pp. 489-498.
15. R. Clark, ed., *UCSD p-System Internal Architecture Guide*, Softech Microsystems, San Diego, 1981.

Appendixes

A: Format of the OPA instructions (hexadecimal)

LOAD SHORT CONSTANT	00..5F
LOAD CONSTANT	6, 0..F* (typespec), value
COMPARE	7, 0..F* (comparecode)
MARK STACK	8, 0..F* (lexlev), 0..D {parmcode}, E..F (funccode)
LOAD VALUE	9, 0..F* (lexlev), 0..F (occurrence), 0..F* (fieldnum)
LOAD ADDRESS	A, 0..F* (lexlev), 0..F* (occurrence), 0..F* (fieldnum)
STORE	B, 0..F* (lexlev), 0..F (occurrence), 0..F* (fieldnum)
SELECT CASE	C, 0..F* (listsize),00..FF(else index), 00..FF{caseindex}
CALL	D, 0..F* (labelindex)
ALLOCATE VAR	E, 0..F* (typespec)
ALLOCATE TYPE	F1, 00..FF(typespec)
LOOP	F2, 00..FF(labelindex)
GOTO	F3, 00..FF(labelindex)
LEAVE	F4
LEAVE CONDITIONAL	F5
UNMARK STACK	F6
ADD	F7
SUBTRACT	F8
MULTIPLY	F9
DIVIDE	FA
MODULO	FB
FORMAL CALL	FC

{ ... } stands for a repetitively occurring field

(...) only explains the parameter to the left of it

F* indicates an escape mechanism:
if the value is 15 decimal then the whole next byte is used instead

16. N. Wirth, "Lilith: A Personal Computer for the Software Engineer," in *Lecture Notes in Computer Science No. 126*, Springer Verlag, Berlin, 1982, pp. 349-397.

17. D. A. Patterson and C. H. Séquin, "A VLSI RISC," *Computer*, Vol. 15, No. 9, Sept. 1982, pp. 8-21.

18. P. Schulthess, "Ein Pascal-System Basierend auf einer Interpretierten Deskriptorarchitektur," dissertation, ETH No. 6099, Swiss Federal Institute of Technology, Zürich, 1978.

19. P. Schulthess and C. Jacobi, "Anatomy of a Small Pascal Compiler," *IEEE Trans. Software Engineering*, Vol. SE-9, No. 2, Mar. 1983, pp. 185-191.

20. P. Schulthess and E. Mumprecht, "Reply to the Case Against Stack Oriented Instruction Sets," *Computer Architecture News* (ACM Sigarch newsletter), Vol. 6, No. 5, Dec. 1977, pp. 24-27.



Peter Schulthess teaches computer architecture at the University of Augsburg, West Germany. From 1980 to 1983 he taught at the Swiss Federal Institute of Technology (ETH) at Zürich. In 1980 he worked for a year in computer network design at the IBM Thomas J. Watson Research Center. His research interests include compilers, broadband local-area networks, and computer architecture.

Schulthess obtained a diploma in electrical engineering from the ETH Zürich in 1969, an MS in electrical engineering from Georgia Tech in 1971, and a doctorate from the ETH Zürich.

Questions about this article can be directed to the author at the Lehrstuhl für Informatik II, Universität Augsburg, Memminger Strasse 6, D-8900 Augsburg, West Germany.

B: Run-time structures

	CONTAINING	SIZE
DESCRIPTOR STACK	DATA DESCRIPTORS	64 BITS
DISPLAY TABLE	POINTERS TO LEXICAL LEVEL	16 BITS
DATA STACK	ARRAYS, RECORDS, STRINGS	8 BITS
BLOCK STACK	BURIED STACK POINTERS, LEVEL INDEX	48 BITS
RETURN STACK	PROCEDURE- AND LOOP RETURN ADDRESSES	24 BITS
TYPE TABLE	TYPE DESCRIPTORS	48 BITS
CODE SEGMENTS	INSTRUCTIONS	64K BITS
LABEL TABLES	PROCEDURE- AND LOOP ADDRESS	24 BITS

C: The stepping mechanism through the type table

```
PROCEDURE describe;
(* performs a typetable walk and leaves a description of the  *)
(* variable in  elemtype, eltsize, name.addr.                 *)

BEGIN endtype  := name.typeinx;
      begintype:= endtype+fieldnum;
      typeentry:= typetable[ begintype ];
      elemtype := typeentry.typeinx;
      eltsize  := typetable[ elemtype  ].size;

      REPEAT (* one step *)

          IF typeentry.link>127

             THEN (* array *) BEGIN
                  indexpop;
                  boundcheck( typeentry, index );
                  step   := eltsize*(index-typeentry.lbnd)
             END

             ELSE (* record *) BEGIN
                  step   := typeentry.offset;
                  eltsize:= typeentry.size
             END;

             name.addr := name.addr+step ;
             begintype := begintype- typeentry.link

      UNTIL endtype=begintype

END (* describe *);
```

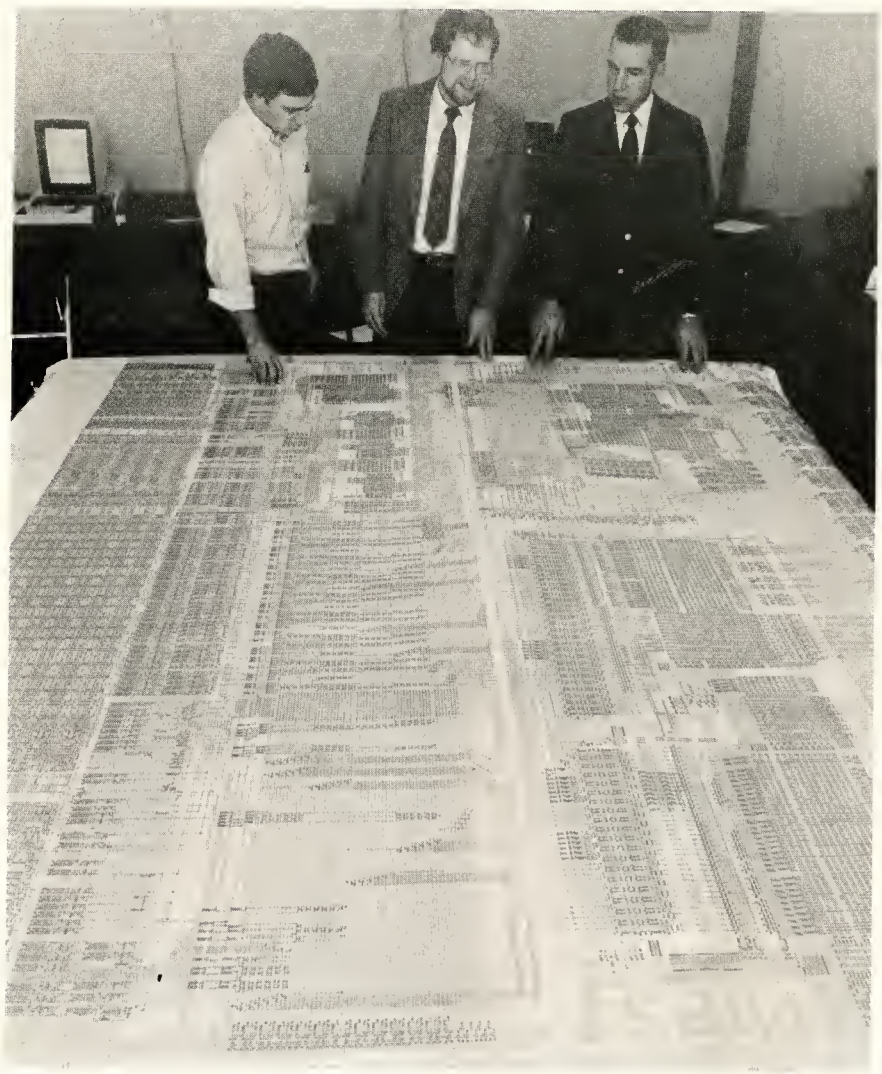
D: Code generation pattern for some Pascal program fragments (no checks, short local addressing)

OPA CODE*	UCSD PASCAL[15]	M CODE[16]	IBM 370[4]
		ch:= 0 ;	
LDSHC 0 ST 0,occ,11	SLDC 0 STL occ	LIx 0 SLWx occ	XR R10,R10 ST R10,occ,11
= 3 bytes	= 3 bytes	= 2 bytes	= 6 bytes
		ch:= ' ';	
LDSHC 32 ST 0,occ,11	SLDC 32 STL occ	LIB 32 SLWx occ	LA R10,32 ST R10,occ,11
= 3 bytes	= 3 bytes	= 3 bytes	= 8 bytes
		A.f3:= i;	
LODV 0,occ(i),11 ST 3,occ(A),11	SLDL occ(A) IND 4 SLDW occ(i) STO	LLWx occ(A) LLWx occ(i) SSWx 4	L R10,occ(A),11 ST R10,occ(A+8),
= 4 bytes	= 5 bytes	= 3 bytes	= 8 bytes
		proc2(i,3);	
LODV 0,occ(i),11 LDSHC 3 CALL proc2	SLDW occ(i) SLDC 3 CLP p2	LLWx occ(i) LI3 CL. p2	L R10,occ(i),11 ST R10,5,R8 LA R10,3 ST R10,6,R8 BALR R9,R1 ; p1
= 4 bytes	= 4 bytes	= 3 bytes	= 20 bytes
		IF ch<' ' THEN proc1;	
LODV 0,occ(ch),11 LDSHC 32 CMPR LESS SLCT 2,p1,255	SLDL occ(ch) SLDC 32 LESI FJP else CLP p1 *else	LLWx occ(ch) LIB 32 ULSS JPFC else CLx p1 *else	L R10,occ(ch),11 C R10,(' ') BGEQ else,R14 BALR R9,R1 ; p1 *else
= 7 bytes	= 7 bytes	= 7 bytes	= 16 bytes
		CASE i OF 0,2,4,6: proc1; 1,3,5,7: proc3 END;	
LODV 0,occ(i),11 SLCT 10, 255, p1,p3 p1,p3 p1,p3 p1,p3	LSDL occ(i) XJP 0,7,else c1,c2, c1,c2, c1,c2, c1,c2,	LLWx occ(i) ENTC size, lo,hi, c1,c2, c1,c2, c1,c2, c1,c2	L R10,occ(i),1 B ,jtab *c1 BALR 9,1 ; p1 B out *c2 BALR 9,1 ; p3 B out
	*c1 CLP p1 UJP out *c2 CLP p3 UJP out	*c1 CLx p1 EXC *c2 CLx p3 EXC	*jtab SLL 10,2 B else B c1 B c2 B c1 B c2 B c1 B c2 B c1 B c2
= 10 bytes	= 31 bytes	= 23 bytes	= 64 bytes

microNEWS

Computer scientist forecasts significantly faster and cheaper computer systems by the end of the decade

By the end of the 1980s, computers will be cheaper and faster, will contain chips that can hold millions of transistors, and will have memory capabilities that outdistance by a factor of 10 the systems presently used, according to John Hennessy, associate professor of electrical engineering at Stanford University. "People will be able to buy machines for $3000-$4000 that are faster than machines that now cost $200,000," he predicts.



John Hennessy (center) of Stanford headed the project that developed MIPS, a high-performance microprocessor requiring only 25,000 transistors, to implement a full 32-bit architecture. Hennessy, who was a member of *IEEE Micro*'s first editorial board, is now working on further performance improvements for MIPS.

Personal computers will be considerably more powerful and able to perform many tasks that are currently associated with minicomputers, while the bigger machines available to scientists and engineers will become significantly faster—"as fast as a Cray," says Hennessy.

"The smaller machines already are rivaling the larger computers. There will be more computer power available for less cost, and people are going to see an order-of-magnitude increase in computer speed. Very low cost computers will be made, consisting of only a few chips, much as the digital watches of today use a single chip. The selling price of such computers will be determined by the physical packaging, not the electronics," Hennessy notes. "Very large scale integration will allow the cost-effective construction of special-purpose hardware for functions such as speech interpretation, graphics, and robotics."

Hennessy is one of eleven Stanford science and engineering faculty recently named presidential investigators by the National Science Foundation, an honor that can bring up to $500,000 of research funds for each recipient. Hennessy is also project director for MIPS—Microprocessor without Interlocked Pipe Stages—the new 32-bit microprocessor that was completely designed and fabricated at Stanford.

MIPS represents a new architecture, housing only 25,000 transistors as compared with the 100,000 or more transistors usually needed in 32-bit machines. Its real strength, says Hennessy, is that it outperforms commercially available microprocessors by about a factor of five. Hennessy is trying to increase the performance of MIPS "by a factor of five to 10," which would make it faster than any existing machine—except for today's supercomputers.

The increased speed was achieved on MIPS by

- improving the compiler technology that translates languages such as Fortran or Pascal into computer instructions;
- redesigning the architecture so that it would be more compatible with the compiler and integrated-circuit technology; and
- devising a chip that relies less on read-only memory (ROM) and instead uses as many transistors as possible to increase instruction execution and speed.

Some commercial chips now contain as many as 450,000 transistors, but 80 percent of these are in ROM. MIPS has 25,000 transistors, but only 15 percent of them are in ROM. This allows for a much faster system, with more chip area devoted to active operation.

The system is designed to let the compiler do most of the translation work, a feature that commercial systems do not have. Most systems do the instruction interpretation as the program is executing. "The instructions MIPS receives are very simple, which means the machine can function faster. VLSI systems produced by Intel or Motorola have more complicated instructions, and it takes longer to execute them," Hennessy explains.

In order to produce a new system that is faster by a factor of five or 10, he is studying the current design of the system to see where the bottlenecks are. In an effort to develop a new architecture for the 32-bit microprocessor, he is working closely with Stanford's Integrated Circuits Laboratory to incorporate any new technology that may develop there. The new chip will have between 100,000 and 150,000 transistors, 50 percent of which may be in memory, according to Hennessy. "We also are trying to determine whether we should supply support for such things as multiplication and floating-point operations on or off the chip," he notes. The more data that can be stored on the chip itself, the faster the chip will be. If information is stored on an adjacent chip, it takes considerably more time to retrieve it.

Hennessy and his associates received a three-year, $5.7 million grant from the Defense Advanced Research Projects Agency in June 1983 to fund their research.

New Editorial Board members announced

Two new members of the *IEEE Micro* Editorial Board have been announced by Editor-in-Chief Peter Rony. They are James J. Farrell III of Motorola in Austin, Texas, and Richard H. Stern, an attorney in Washington, DC.

Farrell is manager of technical communications for Motorola MOS Integrated Circuits Group. Before coming to Motorola, he spent 10 years with Electronic Associates, Inc., working in analog/hybrid computer development. Earlier, he worked on hybrid ICs and power transistor applications for Bendix Semiconductor. He has published widely in the US and abroad.

Farrell graduated from the US Armed Forces Institute in Tokyo.

Stern specializes in the fields of intellectual property and antitrust law. He was formerly chief of the Intellectual Property Section of the Antitrust Division, US Department of Justice, where he was responsible for preparing the government's submissions to the Supreme Court in a number of cases involving the patentability of computer programs.



Richard H. Stern



James J. Farrell III

He received a BSEE from Columbia University in 1954 and an LLB from Yale Law School in 1959.

Competition heats up in microsoftware operating system arena

International Data Corporation's recently published report, *Trends in Micro Software Operating Systems,* reveals keen competitive struggles ahead in the microsoftware operating systems market.

Though the market for transportable operating systems on microcomputers has been dominated by Digital Research's CP/M in the 8-bit environment and by Microsoft's MS-DOS in the 16-bit environment, recent hardware technological advances, changing user demands, and new players have altered the picture dramatically.

In recent announcements, both AT&T and IBM have declared intentions to compete more directly in the microsoftware operating systems field, and now Unix looms in importance. AT&T's move to take stronger control of the licensing and marketing of Unix and Unix-related products, especially for personal computers, and IBM's announcement that it will support its own version of Unix, has strengthened the potential for Unix penetration in the PC environment.

The IDC report focuses on the achievements thus far of the major vendors in creating the generic micro operating systems market, in fostering the portability of software, and in aiding the leveraging of resources committed to the acquisition and development of applications software by both end users and software developers alike.

In 62 pages, the report

- describes the evolution of micro operating systems from the perspective of the current dominant players, Digital Research and Microsoft;

cont'd on page 71

microLAW

by Richard H. Stern
Law Offices of Richard H. Stern
2101 L Street NW, Suite 800
Washington, DC 20037

Chip copyright protection

Both houses of Congress have determined that semiconductor chips should be protected against "chip piracy," but they have selected different preferred means of doing so. The Senate has passed S. 1201, the Semiconductor Chip Protection Act of 1984, an amendment to the federal Copyright Act (17 U.S. Code, Chapters 1-8). The House opted for H.R. 5525, a separate, self-contained law that is outside the Copyright Act, although it is in many ways similar to copyright law. A conference between Senate and House lawmakers will now attempt to reconcile the two bills. It is anticipated that a compromise version of the legislation will be hammered out and enacted into law later this year.

The Senate and House chip bills are both directed to "mask works," a concept they each use that refers to the shapes, patterns, or "topography" of the various layers of a chip. Both bills protect mask works for 10 years and give the owner the right to exclude others from reproducing the work (in a database tape, mask, or ship); selling or importing chips embodying the work; and inducing others to do the foregoing. (The House bill does the latter explicitly, and the Senate bill does so by implication.) Both bills expressly immunize reverse engineering and make one or another special provision for innocent infringement.

As originally introduced, the Senate and House bills both had compulsory licensing provisions, but neither does any longer. The term "compulsory licensing" alarmed many companies, largely ones outside the semiconductor industry. Therefore, the compulsory licensing provisions were eliminated from each bill and provisions were substituted that limited the liability of so-called "innocent infringers" to payment of a reasonable royalty as to infringing chips. The House bill permits a purchaser, such as an equipment manufacturer, to use up all chips that he purchased before he learned that the chip was protected under the Act. But once that inventory is used up the manufacturer can no longer utilize the chip (unless he purchases it from the mask work owner or his licensees). The innocent purchaser is not liable for chips sold (for example, as part of equipment) before the purchaser learned of the mask work owner's rights, but he must pay a reasonable royalty on chips he so utilizes after he acquired notice of the mask work rights. The Senate bill goes further in the purchaser's favor. It allows him to continue to use such chips in his equipment on a reasonable royalty basis even after his existing inventory is exhausted, if the "equities" of the case favor him —for example, if the equipment is a microcomputer laboriously designed around that particular chip. Moreover, the Senate bill allows such a purchaser to make or buy the chip in the future, again on a reasonable royalty basis, if the mask work owner and his licensees cannot or will not supply the chip at a reasonable price. It is understood that the House version is the more likely to be enacted, insofar as this issue is concerned

"Mask works"—the shapes, patterns, or topography of the various layers of a chip—are now protected by federal law.

Under both bills, the similarity between two chips must be quite close for the second chip to be considered an infringement of the first. Both bills contemplate extensive use of expert testimony in determining whether there is an infringement. Both contemplate that functionally dictated chip configurations shall not be protectable. Both suggest, also, that "cells" within chips may be protectable.

Antitrust exemption for R&D joint ventures

Another piece of pending of legislation of interest to the high-technology community is the antitrust exemption measure for R&D joint ventures. Both Senate and House bills have as their centerpiece a declaration that R&D joint ventures shall not be considered "illegal per se" under federal or state antitrust laws. Exactly what this declaration means is unclear, particularly since no judicial decisions have thus far stigmatized any R&D joint ventures as illegal per se. However, the many supporters of the legislation believe that R&D joint ventures will be encouraged by a signal of congressional approval— so that the legislation will serve an important symbolic function. Another feature of the legislation is to eliminate treble damage liability in antitrust suits brought against the joint venture, and to permit injured parties to recover only single, actual damages. Probably, this limitation on liability will apply only to joint ventures disclosed to the government, but this issue is still open.

ROM-less Apple clones

In an interesting pair of developments, the U.S. International Trade Commission and the Treasury Department's Customs Service have reached opposite conclusions as to their respective powers to exclude from import or entry into the United States ROM-less "Apple clone" microcomputers, such as the Orange or Pineap-

ple. Far East manufacturers of these microcomputers have taken to shipping them to the United States without ROMs, to avoid having them barred from entry into the United States on the ground that they contain object code copies of software protected under Apple's copyrights on its operating system software. Once the microcomputers are in the United States, they are equipped with the necessary ROMs and are sold.

Apple protested to both the ITC and Customs. The ITC has power, under Section 337 of the Tariff Act of 1930, to issue an order (to be enforced at all ports of entry by Customs) barring the import of goods involved in an "unfair act or practice" in foreign trade. Customs has independent power, under Section 602 of the Copyright Act, to bar entry of "piratical copies" of copyrighted works.

The ITC concluded that supplying ROM-less microcomputers, which would later be combined with ROMs in the United States to complete the product and cause a copyright infringement, was "unfair." But Customs ruled that a piratical copy is a *copy*, not something that will be used with a copy, and it therefore declined to issue its own order barring entry of the "Apple clone" ROM-less microcomputers. However, Customs will enforce the ITC's order.

Hearing record available

The record of the Hearing on S.1201, the Semiconductor Chip Protection Act of 1983, before the Subcommittee on Patents, Copyrights, and Trademarks of the Senate Committee on the Judiciary (May 19, 1983), is now available. The 168-page paperbound volume includes testimony and statements from Representative Don Edwards of California, Assistant Register of Copyrights for Legal Affairs Dorothy Schrader, Thomas Dunlap of Intel Corp., Dr. Christopher Layton of Intersil Inc., Harvard Law School Professor Arthur Miller, Jack Biddle of the Computer and Communications Industry Association, Ronald Palenski of the Association of Data Processing Service Organizations, Patent Office officials, and Leslie Vadasz, an IEEE Fellow and official of Intel Corp. *IEEE Micro* readers interested in receiving a complimentary copy can request it from Senator Charles Mathias, U.S Senate, Washington, D.C. 20510; enclosure of a self-addressed mailing label will facilitate response.

microNEWS *cont'd from page 69*

- outlines the overall strategy of these two major developers;
- presents other product strategies where they enlarge upon and/or contribute to the company's operating system strategy;
- discusses the recent spate of Unix-related announcements; and
- examines up-and-comers such as AT&T, Softech Microsystems, and Pick.

The report concentrates on detailing the current status, strategies, and future plans of the two market leaders and the leading contender. Some highlights follow:

Digital Research. Digital's identity is very much tied into being a partner in software solutions with both applications developers and end users. In all, the company has over 1000 OEM accounts (about 500 for all operating systems), and their products represent about 1.5 million licensed users.

Microsoft. Though the company is not subservient to IBM, it is undeniable that its association with the PC has been fortunate. It has been estimated that about 40 percent of all the personal computers shipped in 1983 were shipped with the Microsoft operating system. There are also estimates that MS-DOS "owns" about 90 percent of the 8088/8086-based operating systems market.

Unix—AT&T Technologies. Unix, by far the most sophisticated of the operating systems discussed, has had the broadest migration of any portable operating system. Some estimates indicate that up to 70 hardware vendors have implemented some form of Unix. Approximately 60,000 source binary licenses have been signed, and about 100,000 Unix machines are currently installed.

Other companies briefly profiled in the report are Softech Microsystems Inc. and Pick Systems.

The report is available through IDC's Software and Services Information Program for $1800. For further information, contact Damian Rinaldi at (617) 872-8200.

Apple Computer calls Macintosh a success

Apple Computer announced it has shipped more than 70,000 Macintosh computers since the product was introduced on January 24. The company originally projected that it would ship 50,000 units within the first 100 days, which ended May 3, and reached that quantity in just 74 days. Apple expects to sell 250,000 Macintosh computers by September 28, the end of its 1984 fiscal year.

The sales forecast was part of the criteria established by Apple to measure the success of the Macintosh computer during its first 100 days on the market. Apple also predicted that the Macintosh would emerge as a milestone product among consumers, computer dealers, and independent software developers.

"Apple set very aggressive goals, and we believe that we have exceeded those goals. Based on the response we've had so far, we feel that the Macintosh is on its way toward becoming the third industry standard product in the personal computer business," said Steven P. Jobs, chairman of the board at Apple. "We reached our 50,000 unit sales goal in just 74 days. By comparison, it took two and a half years to sell 50,000 Apple IIs, and seven months to sell the same number of IBM PCs. Consumer demand for Macintosh during the first 100 days was at least double the 70,000 units that we could produce."

Response from independent developers and authorized Apple dealers has been equally positive. Since the introduction of the Macintosh, Apple has received over 5,000 inquiries from independent developers. More than 1000 computers have been sold to these firms. Of the 150 software packages anticipated by the end of the year, 50 are expected to be available this summer.

During Macintosh's first 100 days, agreements were reached with three of the nation's largest business-oriented retail chains to sell the computer, indicating significant acceptance of the Macintosh in the business market. Sears Business Systems Centers, Businessland Inc., and The Genra Group—who serve the office, professional, and business markets—all announced plans to carry both the Macintosh and Lisa.

microREVIEW

by David L. Hannum
AT&T Information Systems
Room 59-3A37
99 Jefferson Road
Parsippany, NJ 07054

We have heard much of portable computers in the past few years, and we have seen a trend begun by the Osborne 1 evolve into what is now a varied and stable market in its own right. Most of these "portables," however—weighing in at 26-30 lbs. and packaged in somewhat unwieldly proportions—have earned the more appropriate moniker "transportable." In other words, you have to think about it before you pick it up. You don't want to move a transportable any more than you have to, and there are plenty of places you wouldn't take it.

Last year's introduction of the 9-lb., battery-powered Gavilan computer changed all that. With its flip-top LCD screen and its built-in disk drive and printer, the Gavilan was a commuter's dream. That is, until he saw the price tag: $4000 without the printer.

Although its price banished it to the executive toy category, the Gavilan did redefine industry standards for the portable micro. Since then, two entries into the "mini-portable" class by major manufacturers—Hewlett-Packard and Sharp Electronics—suggest that this trend, too, may catch on. HP's new Portable 110 is still expensive at $3000, but it may provide interesting material for a review at a later time.

The Sharp PC-5000

This month we'll look at the PC-5000 by Sharp, a mini-portable that does just about everything the Gavilan does for about half the price.

THE SHARP PC-5000 is a good representation of the new industry standard for truly portable microcomputers: small, light, battery-powered and fully equipped, this model is designed both for travel and for use in transit. The system currently comes bundled with the Sorcim family of software, and other popular software programs are available as well.

Sharp has managed to incorporate not one but *two* methods of software storage into the PC-5000: internal floppy disc drives *and* the first standard use of bubble memory. The computer operates on a rechargeable lead battery, and with the optional internal printer and modem, it still weighs in at under 13 lbs.

Responding to a need by professionals for a complete take-anywhere, anytime system, Sharp seems to have filled the bill. Now if Sharp USA can support this product with the service and dealer support it deserves, it may get a foothold in the volatile personal computer market.

The basic PC-5000 is a single unit with a standard QWERTY keyboard, eight function keys, and flip-top LCD screen with an eight-line by 80-character matrix. The heart of the computer is its 16-bit Intel 8088 CPU, which uses the MS-DOS operating system.

The PC-5000 is said to be compatible with the IBM PC, but because of the changes necessary to accommodate the smaller display and the fewer function keys (eight instead of 10), not one of my IBM PC programs would run on this unit. This may be overcome with a little creativity on the user's part. It will probably not pose a major problem anyway, since there's already a substantial amount of software for the machine (more on this later).

The PC-5000 is the only mini-portable which currently offers an optional printer that is internal to the unit (not a clip-on), and it is a very clever device indeed. The virtually silent "thermal/thermal transfer" printer provides the user with a choice of printing methods: (1) thermal printing using heat-sensitive thermal paper, or (2) "thermal transfer" printing, which uses a disposable ribbon cartridge to produce near-letter-quality type on plain bond.

Printing is not very fast (37 characters per second), and the type is fairly small. On the whole, though, I would say that its output is more than satisfactory for on-the-spot hard copy—and it does have graphics capabilities as well.

The most interesting aspect of this machine may be its many methods of memory storage. Built into the box is 128K of RAM (expandable to 256K) and 192K of ROM. Internal floppy-disk drives are also available in either the 5¼-inch or 3½-inch formats, both with 360K capacity. The disk drives are expensive, though: $995 for the 5¼-inch version and $699 for the 3½-inch.

Bubble memory

But the real star of the Sharp PC-5000 is its bubble memory storage for software, which constitutes the first standard use of the bubble technology in a computer of this type.

The machine comes standard with one 128K bubble cartridge, but additional bubble units can be purchased for $269 each. Although a limited amount of software is currently available on the bubble units, I am told that many dealers will download your favorite programs onto cartridges for no extra charge.

The computer is powered by a rechargeable lead battery which—according to the marketing rep I spoke with—provides for eight hours of constant use without the printer and 1½ to two hours of use with the printer in constant operation.

The unit still weighs in at only 10 lbs. unless you add the optional 300-baud modem, which brings it up to about 13 lbs.

Software for the PC-5000

The system currently comes bundled with Sorcim Superwriter word processing; Superplanner, an executive planner/calendar program; Supercalc; Supercomm, a communications package; and Supertools, a set of utilities integrated into the menu scheme that allows easy use of Dir, Copy, and other important functions.

Also available are the popular Wordstar, dBase II, PFS:File, PFS:Report, Easyplanner, Easywriter II, and Easycomm. The marketing rep assures me that more are on the way.

I had no trouble adapting to these programs; in general, they performed every bit as well as they were supposed to. In fact, I was pleasantly surprised by an occasional bit of user-friendliness—such as the "autoboot" program, which brings up the menu screen and applies a few human-engineered touches to the PC-5000 software.

Sharp has pulled off a neat trick in packing the latest "stable" technology into a machine this small at such a reasonable price. My only real complaint about the PC-5000 is the glare (and thus eventual eyestrain) produced by the LCD screen. Otherwise, Sharp's competition may have a hard time keeping up with it. At $1995, this machine may just be a mini-portable "for the rest of us."

Keep those cards and letters coming. Your requests and comments are much appreciated, and I will try to respond to each of them as time permits.

SYSTEM AT A GLANCE

Sharp PC-5000 by Sharp Electronics Corporation
100 Sharp Plaza
Paramus, New Jersey 07652
(201) 265-5600
1-800-732-8321

GENERAL

Model	PC-5000
Dimensions	12-27/32 × 12 × 3-7/16 inches
Weight	10 lbs.
Price	$1995 (basic unit with one bubble unit and no disk drives)

HARDWARE

Processor	Intel 8088
Word size	16 bits
Memory	128K RAM, expandable to 256K 128K bubble cartridges (optional) 192K ROM
Floppy disks	5.25-inch disk drive, 360K bytes each (one or two; optional)
Display	LCD, eight lines × 90 columns Graphics: 640 × 80 dots
Interfaces	RS-232C port (one) External bus port (one) Audio cassette port (one) RAM/ROM expansion slots (two)
Printer	Built-in: thermal or ribbon (optional) 37 characters per second, 80-character width, also prints graphics
Keyboard	Standard, with eight function keys
Modem	Optional, 300-baud, 103-compatible; can be used as a conference phone
Power	Internal rechargeable lead battery or recharger pack

SOFTWARE

Operating system	MS-DOS 2.0
Languages	Basic
Word processor	Superwriter
Data manager	Superplanner
Spreadsheet	Supercalc^2
Communications package	Supercomm
Utilities	Supertools



MANUALS AND TUTORIALS

Manuals	Provided for each software package. A user's manual is provided with both the PC-5000 and each optional hardware unit.

microSTANDARDS

by Robert G. Stewart
Stewart Research Enterprises
1658 Belvoir Drive
Los Altos, CA 94022

Is Unix suited to be a standard for micros?

Recently I've been using Unix III on a Motorola 68000/IEEE 696-based system to perform computer graphics analyses. And after reading AT&T's ads suggesting that Unix is our prime candidate for a microcomputer operating system, I'd like to share with you a few conclusions based on my personal experience with it.

Let me begin by noting that the first two letters of Unix are the same as the first two letters of "unfriendly." The unnecessarily abstruse language comprising the command structure seems intended primarily to reduce keystrokes, as if everyone were a one-finger hunt-and-peck typist. Next, one confronts a directory and file secrecy system clearly designed for a large computer installation rather than a single-user microcomputer. Multi-user operation with Unix is a real measure of the capabilities of a disk drive, since swapping to disk occurs frequently. If you have anything less than a high-performance hard-disk drive, the multi-user performance is totally inadequate.

A true and utter disaster is the absence of a file back-up system for the editor, Vi. The user must devise his own backup methods to avoid calamity in case of a hardware crash. Still, even with this precaution, it is easy to lose all the backups if the hardware is at all flaky. Unix prefers to keep files in RAM, rather than writing them out to disk after an edit is exited. To avoid file loss, multiple "syncs" must be used before and after edit, load, and compile operations to force the operating system to write the files out to disk.

In many applications, microprocessors must interface with hardware. This can require fixing and accessing code in certain memory locations. A multi-user, memory-mapped operating system objects to such usurpation of its prerogatives. Only a "superuser" has access to this privilege in Unix. This once again shows the multi-user limitations of Unix for applications involving direct hardware interfacing.

Having used Gary Kildall's CP/M operating system for eight years, I really appreciate finding space left in memory for my programs. Not only does Unix need about 15 times as much RAM as CP/M, it needs more than 10 megabytes of disk space to contain the system. So. . . is it worth it? In my opinion, for a single-user microcomputer system, probably not. The years of lily gilding which have gone into Unix are overkill for this type of application.

Well then, what's good about Unix? Clearly the massive documentation which is supplied or available is a real credit to Bell Labs' determination to provide users and system maintainers with credible writeups of myriad details. The screen editor, Vi, is reasonably powerful and easy to use, with the exception of the backup faux pas mentioned above. The program fsck (file system check) allows salvaging munged disks. Shell script files and alias files help make Unix somewhat more user-friendly.

The programming language C is overly compact and seems designed to minimize writing lots of comments. The reason is that the code is so dense, with several statements typically contained on one line, that comments are hard to include with the same density. In this sense C reminds me of Forth. Further, the subroutine response to a calling program is different from what is expected by, say, Fortran. As a result, at least for Silicon Valley Software's Fortran 77 compiler, "wrappers" are required to interface C and Fortran routines. Moreover, this compiler has no capability for bit manipulation in bytes and words—a severe handicap for hardware interfacing. Evidently, high-level-language compiler writers do not understand that systems that talk to hardware need direct bit-manipulation capability.

To touch on yet another point, the hardware problems of megabyte memory cards made me a believer in the necessity of error correction, rather than simple error detection, for large memory systems.

Unix needs to be made simpler, safer, cheaper, faster, and more user-friendly. It also needs to provide better hardware interfacing and significantly expanded graphics capabilities.

Dennis Blake

From Paul Borrill, chairman of the P896 working group

It is with great sadness and regret that I inform you of the recent death of Dennis V. Blake, BSc, C Eng, FIEE.

Dennis Blake was known to us all for his work in editing the P896 specification. He successfully created a great document from the sometimes haphazard and inadequate inputs we provided to him. Dennis spent a great deal of his professional life working with the standards and frequently represented the UK in international standards meetings. He was also instrumental in obtaining funds from the UK Department of Industry to allow individuals to participate fully in the development of the P896 standard. Dennis was the project officer appointed by the Department of Industry to oversee the use of these funds as well as chairman of the IEE's Subcommittee on Computing Standards—the parent body responsible for the IEE Working Party on Backplane Buses.

Without Dennis' considerable efforts and strong personal commitment to the working group, the P896 specification would not have been completed in the time it was. Indeed, it might not have been completed at all.

I am sure I speak for all members of the working group when I say that Dennis will be sadly missed from our meetings. For many of us, Dennis' departure means saying goodbye to a dear and valued friend.

To obtain more information about a product or service featured in New Products, circle the appropriate number on the Reader Service Card at the back of the magazine.

NEW PRODUCTS

New Products Editor: Victor P. Nelson

Logic simulation system offered for IBM PC/XT

Logicpro, a software package that converts any IBM PC/XT into an engineering workstation for logic design, has been introduced by E/Z CAD. Designed for both board-level and integrated-circuit logic designers, Logicpro offers an alternative to high-priced workstation packages.

With capabilities comparable to mini and mainframe level industry standard simulators such as Tegas and Logis, E/Z CAD's system fits application areas ranging from gate-arrays and custom and standard-cell users to system designers using 7400 or 4000 series logic families.

Logicpro, semi-interactive professional logic simulation system, includes several packages. One is Waveform, a high-resolution graphics utility, that displays the timing diagrams generated by Logicpro in color. Also useable with Logicpro are the TTLLIB and CMOSLIB libraries, each containing over a hundred 7400 and 4000 series device models.

Logicpro can simulate all standard logic elements, including three-state elements, n-and p-channel transistors, CMOS transfer gates, wired-or nodes and RAM and ROMs with its built in functional models. It can also perform fault-detection for stuck-at-one and stuck-at-zero nodes, automatic spike detection and race analysis, realistic timing analysis with both rise and fall propagation delays and DC initialization analysis.

Logicpro permits easy troubleshooting of designs with features such as user control of DC convergence algorithms, ability to output all or partial network states at any time step during simulation, and network listing and verification tables.

Logicpro is available immediately in two versions: Logicpro/3000 with 3000 nodes capacity needs 320K system and is priced at $650. Logicpro/1000 with 1000 nodes capacity needs 256K system and is priced at $495. The Waveform utility, at $350, requires a color graphics board and monitor or shades-of-green as in COMPAQ. The TTLLIB and CMOSLIB at $150 each are part-number-based and pin-referenced for ease of use and are user expandable. The package comes with a complete manual including a tutorial and examples.

Reader Service Number 1

Minicomputer features fault management

The Parallel 300, a minicomputer that incorporates fault tolerance, has been introduced by Parallel Computers.

The Parallel 300 is intended for use in operational information-systems applications, those environments where computers are critical to the flow of products and services. Typical applications could include factory automation; office automation and engineering workstation networks; communications industries; car-rental agencies, travel agents, hotels, and service businesses that use on-line computers; trucking, freight handling, and shipping; and medical clinical labs.

Parallel 300 computer users need no training or tools to restore the system to its fully operating state. All components can be removed and replaced quickly without tools, usually within a few minutes. In the event of a component failure, the computer alerts the user to the exact nature of the problem and provides step-by-step instructions for replacing the faulty component. Parallel also maintains a 24-hour, seven-day-a-week hotline to assist users.

The Parallel 300's redundant architecture duplicates all vital components, including processors, disk drives, disk controllers, and power supplies. If any of these components fail, the computer protects itself from data loss, error, or shutdown, and continues to operate.

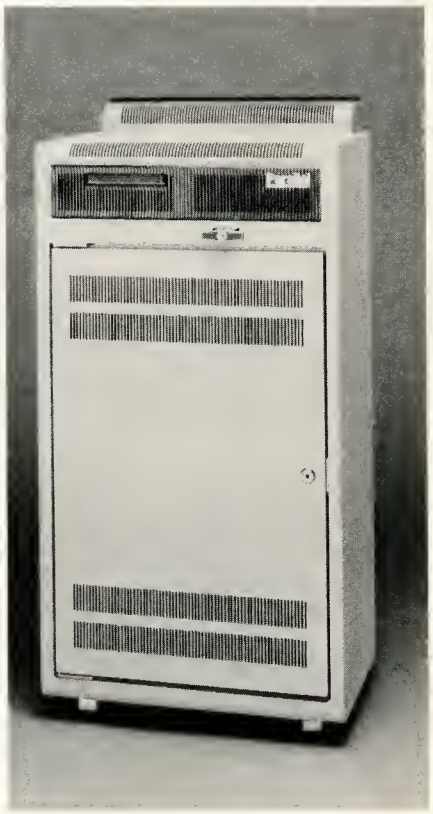
A closely coupled pair of parallel-processing units, each using a Motorola MC68010 virtual-memory microprocessor, executes all tasks simultaneously. If one processor fails, the breakdown is detected by the Parallel 300's synchronization logic. A self-diagnostic procedure identifies and isolates the faulty processor, and the remaining processor continues to run the application.

Also duplicated are Winchester disk drives and disk controllers. All data are written onto both disks, and each disk is connected to a separate controller. If a controller fails, the system isolates it while the other continues alone until the component can be replaced. The redundant disk architecture also provides for motor malfunctions such as a read or write error stemming from a defect in the disk media.

Parallel 300's entire system runs directly off its batteries, which are kept charged by external power. The use of batteries effectively isolates the system from fluctuations in external voltage. In case of a total power failure, the batteries keep the system operating for up to 15 minutes. An optional battery pack maintains the system for up to 90 minutes. In the event of extended power loss, the system will shut itself down systematically while safeguarding data. When external power resumes, the system will automatically continue processing from the point it left off.

The Parallel 300, Model 30 has a list price of $74,900. Volume discounts are offered to OEMs. Optional expansions to the basic system include additional memory, disk, communications, local area network, and software.

Reader Service Number 2



Three key support chips enhance microprocessor system

Intel announced two new controller chips, the 8208 and 82062, and an improved version of another key microprocessor system support chip, the 8256AH. The company claims that these three products make it easier for engineers to design main memory, mass storage, and communications subsystems. Intel's 8208 high-speed dynamic RAM controller supports faster design of dynamic RAM memory by combining the control functions onto a single chip. The 8208 simplifies the dynamic memory interface and refresh requirements to permit designers to add larger amounts of dynamic RAM without trading off shorter design cycles and reliability. A single-chip controller, the 82062, provides nearly all the logic needed to build a Winchester disk drive controller subsystem.

The 8256 multifunction peripheral chip has been improved to offer higher speed and a lower cost plastic packaged part, the 8256AH. Reduced in area from 220 mils per side to 159 mils per side, the 8256AH combines the functions of four individual chips to create a programmable serial communications interface, parallel I/O ports, five programmable timer/counters and an eight-level priority interrupt controller.

The 8208 replaces up to 20 TTL chips or 9 MSI chips. It provides all the signals needed to control 64- and 256-kilobit dynamic RAMs. This single VLSI chip directly addresses and drives up to 1M byte of memory without external drivers. Designed for the simplest interface to Intel's 80186 and 80188 microprocessors, the 8208 can also be used with other microprocessors.

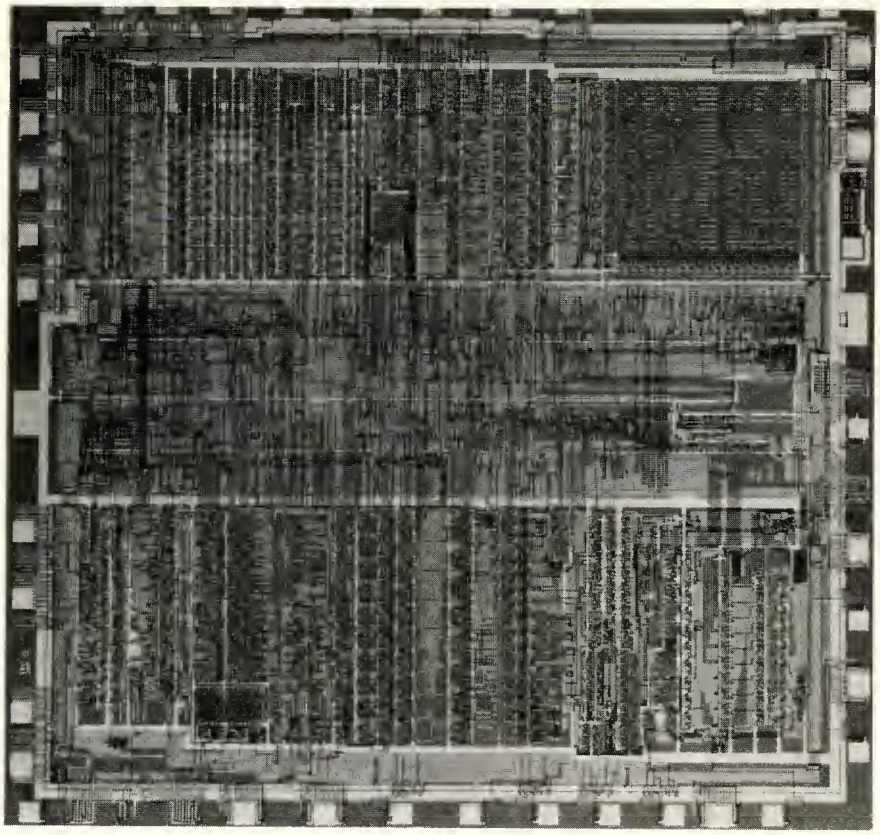
The 82062 is designed for personal computer and office workstation designers whose systems will use the new 5¼-inch disk drives from manufacturers such as Seagate, Shugart, Tandon, and Computer Memories.

The 82062 provides all the drive-control logic plus the control signals that simplify the design of external phase-locked loop and write-precompensation circuitry. Controlled by the system's CPU using six commands, the 82062's current transfer rate is 5 megabits per second. Future versions will support the new 10 megabits-per-second ST412HP and ESDI interfaces. Sector lengths can be programmed to be 128, 256, 512 or 1,024 bytes.

By using six high-level commands—restore, seek, read sector, write sector, scan ID and write format—the CPU can control the initialized 82062 with single byte commands. In addition, the 82062 adds all the required track formatting to the data field, including 2 bytes of CRC code.

Intel's 8256AH multifunction peripheral chip combines four commonly used functions into a single 40-pin device. It supports serial communications, parallel input/output, timing and event counting and priority interrupt control. All of these functions are fully programmable, based on the contents loaded into its nine registers. In addition, the five timer/counters and two parallel I/O ports may be accessed directly by the microprocessor.

This improved HMOS II version of the Siemens 8256A and the Intel 8256A has the same on-chip, baud-rate generator, a 1 megabit-per-second serial port, two 8-bit parallel I/O pots, five 8-bit programmable timer/counters (four of which can be cascaded to create two 16-bit timer/counters) and an 8-level interrupt controller. The new plastic packaged 8256AH is faster than the 8256A. The 8256AH replaces several lesser integrated components making the overall design smaller and more reliable.



Intel's 82062 Winchester disk controller.

Reader Service Number 3

Emulation tools run almost any CP/M-80 program on a PC without boards or hardware

80Mate, a CP/M-80 emulator utility, and 80Term, a terminal emulation program, are available from Vertex Systems. Most CP/M-80 programs can be run on PC/MS-DOS micros without the need for expensive coprocessor boards. CP/M-80 programs run with 80Mate use existing CP/M commands.

80Mate uses ordinary PC/MS-DOS disks and devices. The unprotected CP/M-80 program is transferred to a DOS disk. 80Mate is run, and the CP/M-80 program is "booted" and operates as though on the computer it was written for. CP/M programs can be transferred using conventional serial file methods. Vertex suggests using the Xeno-Copy Plus format conversion utility when speed, accuracy, and convenience are required. Safe to use with hard disks, 80Mate is priced at $99.50.

Vertex Systems fully supports end-users and dealers with technical help. 80Term, priced at $44.95, may be necessary for several selected programs to match the display characteristics to the new host computer.

Reader Service Number 4

Converter runs CP/M-86 software on PC-DOS, MS-DOS operating systems

The O.S. Converter from Dynamic Microprocessor Associates permits PC-DOS (MS-DOS) object code to run on CP/M-86 microcomputers and enables CP/M-86 object code to run on MS-DOS systems without loss of speed. The O.S. Converter permits users to run such programs as Basic, Fortran, Pascal, and other language compilers, as well as utilities like Assembler and Linker.

The new program operates by loading a target program into memory and creating the environment that the program expects. There is no interpretation of instructions; the program itself remains in control of operations. The O.S. Converter is 4K bytes. When in use, it resides just above the operating system in RAM and enables the program being run to take full advantage of available memory. The O.S. Converter for the IBM PC is supplied with a companion program that enables PC-DOS systems to read CP/M-86 files.

The O.S. Converter for use with either PC-DOS (MS-DOS) or CP/M-86 is available now for $115.

Reader Service Number 5

UNIX port to IBM PC/XT available

International Data Services, a UNIX systems and software supplier, is offering a UNIX port to the IBM PC/XT. The product, UnX-II, is an implemenation of AT&T Technology's UNIX System III with Berkeley 4.1 enhancements. UnX-III adheres to the standards of the AT&T version while offering the performance enhancements of 4.1 BSD.

UnX-II is a full implementation of UNIX System III, making available to the user of the IBM PC/XT the powerful UNIX functions found on minicomputers. Among these features are multitasking and dual-user capabilities. In a two-user system, the PCs or XTs may be connected via a sync port directly saving the expense of a local area network.

UnX-II will be offered in both a cost-effective, bundled package as well as in an unbundled form, allowing users a choice of pricing and application. The basic product, which retails for $595, includes the system III Kernel, basic (Bourne) shell, utilities, and uucp.

Among the optional utilities are the Berkeley 4.1 enhancements, including the VI editor and C shell, at $150. A PC DOS shell, which allows a non-UNIX-familiar user to make full use of the system via PC DOS commands, is available for $150. SCCS is available for $100. These options may be purchased as a package for $395. The UnX-II port plus all options are available in a bundled package for $895. Distributor discounts are available. IDS offers the Writers Workbench at $995.

The IBM PC/XT hardware requirements for full use of the port include a minimum of 320K byte memory and the optional installation of the Intel 8087 math processor for enhanced performance. All of the standard UNIX system calls are supported. A manual is available from IDS for $25 that describes system functions and operation.

Reader Service Number 7

Masterforth available for Z-80 CP/M 2.x users

For Z-80 CP/M 2.x users, MicroMotion is offering Masterforth, the complete professional programming language that includes a built-in macro-assembler with local labels, a screen editor, and a string handling package.

The input and output streams are fully redirectable and use the host operating file structure. Also available for Apple II/II+/IIe users, Masterforth features a floating point option.

The package includes Forth Tools, a 200-page textbook, a technical reference manual, and a complete listing of the Masterforth nucleus. Forth Tools gives the user an in-depth view of input and output, from reading the input stream to writing a mailing list program. Numerous examples are also provided.

Masterforth retails for $100, and with floating point for $140.

Reader Service Number 6

Speech recognition equipment operates over standard telephone lines

Votan now has available the VPC 2000 Voice Card, which provides voice recognition, speech generation, and telephone management capabilities in one package. With Voice Key software, the Voice Card can be used immediately for speech command and program control.

Votan has also introduced the VTR 6000 Voice Terminal. Connecting to any computer supporting an RS-232 serial port and xon/xoff protocol, the Voice Terminal can connect indirectly to the host, or be connected between a CRT terminal and a host to supplement keyboard/screen transactions. Voice Key software permits immediate speech command and control of the computer.

Voice Key software incorporates voice capabilities into existing software packages without any modifications to the application software itself. For each application program, the user may define up to 64 voice utterances that are linked to sequences of key strokes, up to 30 characters each. At runtime, the Voice Card listens for voice commands from the user, automatically typing the key strokes linked to each command when it is uttered. The keyboard remains active and can also be used in a normal fashion. For each utterance, the user may choose whether or not to hear voice response messages to confirm the voice command.

Voice Key at runtime, for the Voice Terminal, listens for voice commands from the user, automatically typing the key strokes linked to each command when it is uttered. For example the spoken word "initialize" might be used to set up normal parameters on a word processing program, such as to set the right margin, double space, turn off justify and hyphenate options, and eliminate the help menu.

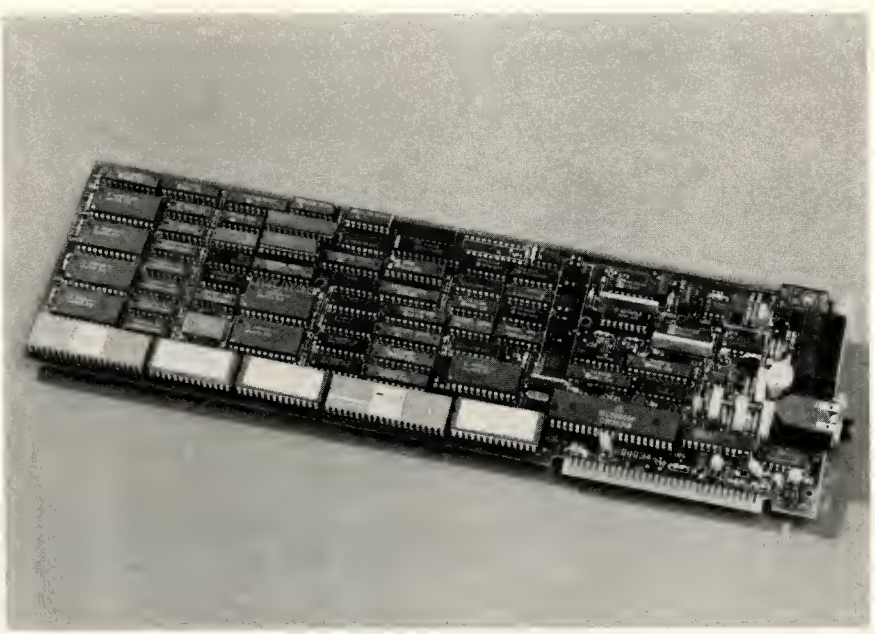
The Voice Card features continuous speaker dependent recognition, which allows an individual to speak to the computer in a normal conversational flow. A word spotting capability will pick out target words located anywhere within a stream of normal conversation.

Voice Card operates over standard telephone lines. Two-way communication with a PC from remote locations is possible. Voice Card provides complete telephone interfacing capabilities including auto answer, auto dial, and touch tone encoding and decoding. A software program, the Telephone Manager, is provided to give users immediate access to these hardware facilities. It provides a voice-controlled telephone dialer and a sophisticated automatic answering/voice mail system.

The VPC 2000 is composed of a printed circuit board that plugs into any of the long auxiliary system bus slots on the IBM PC. Microphone, speaker, and complete software and documentation are included in the $2,450 price.

The VTR 6000 is a stand-alone voice I/O subsystem with its own chassis, power supply, microphone, and speaker. Complete software and documentation are included in the $3,950 list price.

Reader Service Number 8



Designer pop-up menu software and upgraded optical mouse are available

Mouse Systems has announced Designer Pop-up menu software and its second generation optical mouse family.

The new software allows personal computer users to design or personalize pop-up menus for most IBM PC and PC-compatible software. Pop-up menus eliminate the need to memorize and type complex commands. Users simply select a command from the pop-up menu by pointing at it with PC Mouse and pressing one of its three buttons.

Designer Pop-up menus work on the IBM PC, IBM PC-XT, PCjr, and PC-compatibles and require no modifications to existing software. Users can design or personalize their own pop-up menus with any word processor that creates ASCII text files and a compiler provided by Mouse Systems.

Unlike fixed menus, pop-up menus used with the PC Mouse appear on the screen only when needed and disappear when the desired command is selected. The pop-up menus are completely transparent to the underlying program and take up no screen space except when they are actually being used.

For $20 Mouse Systems offers an upgrade to Version 3.00 to current pop-up menu users. The latest version includes pre-configured pop-up menus for Lotus 1-2-3, VisiCalc, Multiplan, Personal Editor, Volkswriter, WordStar, SuperCalc3, PFS:WRITE, and Multimate. Version 3.00 also works with Microsoft Word, the Multi-Tool series, and Window Manager when available.

The list price of Designer Pop-up menu software, Version 3.00, is $95.



Second Generation Mice

Mouse Systems' second generation of optical mice, designated M-2, does not require calibration and includes PC Mouse, FieldMouse, and the basic M-2 Mouse. Other optical mice require calibration each time they are connected to power; this involves moving the mouse in large circles on the pad for several seconds.

PC Mouse is an M-2 optical mouse that includes Deisgner Pop-up menu software. PC Mouse connects to the IBM PC, IBM PC-XT, PCjr, and PC-compatibles through a standard RS-232C serial interface. Software installation is a one-time, one-line procedure that requires no modifications to existing application software.

FieldMouse is an M-2 optical mouse for use with non-IBM computers. It is similar to the PC Mouse, but does not contain pop-up menu software. The basic M-2 optical mouse is available in a variety of protocols and colors to meet the special requirements of OEM customers.

PC Mouse is compatible with PC-DOS Versions 1.10 and 2.00. It also works with Visi On, DesQ, Halo, CADPLAN and Auto CAD.

Reader Service Number 9

Data sentry modem prevents data theft

Data Sentry, an intelligent modem from Lockheed-Georgia, can prevent data theft and other security breaches of personal, mini- and mainframe computer systems without requiring encryptions or changes in systems programming.

Data Sentry uses a sophisticated call-up, call-back, and password sequence to thwart data thieves. It also offers all the telecommunications features of conventional nonsecure intelligent modems.

An optional companion device, Remote-on, can turn a computer's power on and off from a remote terminal once security has been cleared.

Data Sentry's security routine requests the phone number of someone wanting access to its computer. Then it hangs up and looks through its list of authorized numbers. If the caller's number is authorized, Data Sentry dials the caller back and requests a password. The caller has three tries to give the right password. Without it, Data Sentry won't return further calls from that phone number. Data Sentry also creates an audit trail of user passwords and phone numbers, whether calls are successful or not.

A lower security mode, ideal for traveling business people, allows users to program Data Sentry to call back any number from which it gets the correct password. This mode is suitable for doctors, lawyers, and sales representatives who want access to the computer during non-business hours.

With Remote-on, Data Sentry turns the computer on and off. For less restricted but still secure protection, Data Sentry will allow access to the computer system upon entry of only the correct password.



In a nonsecure mode Data Sentry operates as a conventional intelligent modem. It features 300 or 1200 baud, full duplex, asynchronous operations with auto dial, auto answer, auto speed, and auto party selection.

Six programmable directories are available for security and convenience. These range from ten 32-character "not allowed" phone numbers that Data Sentry won't call back when it's in a security mode.

Terminal interface is a standard RS-232C cable. The telephone interface is a single telephone number drop with an RJ-11C connection. A battery backup protects menus and tables in memory during moves and power failures. A forced answer/originate feature is available for use with leased lines.

List price of the Data Sentry is expected to be $895, while the Remote-on option lists for $145.

Reader Service Number 10

ICs combine low power consumption, high performance in data acquisition subsystems

Two 8-bit, serial-output, A/D peripheral integrated circuits that combine low power requirements with fast conversion times are now available from Texas Instruments. The TLC540 and TLC541 can perform analog-to-digital conversions in 10 microseconds and 19 microseconds, respectively. Including microprocessor access, this translates to taking a sample and performing an 8-bit ratiometric A/D conversion up to 71,000 times per second (TLC540) or 28,000 times per second (TLC541). Each product dissipates 6.5 mW.

The TLC540 and TLC541 can be used as complete data-acquisition subsystems providing measurement data to a microprocessor via a serial data port. Possible applications include test equipment, automotive instrumentation, industrial controls, robotics, toys, home computers, environmental controls, and some signal-processing applications.

As serial-output data-acquisition ICs, the TLC540 and TLC541 perform a variety of functions needed to interface analog inputs to microprocessor systems. On-chip features include an 8-bit, successive-approximation A/D converter, a 12-channel multiplexer, 11 analog inputs, sample-and-hold circuitry, three data registers, and microprocessor-control logic interfacing with an on-chip, three-line, TTL-compatible serial I/O port.

The chips provide a built-in self-test mode and the capability for simultaneous read/write operation. Separate system and I/O clocks allow broad control flexibility.

In addition, the sample-and-hold circuitry is software programmable. By allowing designers to control precisely when the chip takes its sample, the TLC540 and TLC541 can be used as highly accurate sampling devices. The TLC540 has a 4-MHz system clock and a 2-MHz data clock (maximum); the TLC541 has a 2-MHz system clock and a 0.5-MHz data clock (maximum). The chips are accurate to +/− ½ least significant bit (LSB) over the entire temperature range.

The TLC540 and TLC541 are pin-for-pin compatible with the MC145040, but offer conversion speeds 3X and 2X faster, respectively. The two parts are available in military (−55 to 125°C) and industrial (−40 to 85°C) temperature ranges. A single 5-V supply is required.

Available now in 20-pin dual-in-line packages, the 100-piece prices of TLC540 for industrial and military uses are $3.60 and $7.34, respectively, the 100-piece prices of the TLC540 for industrial and military uses are $3.60 and $7.34, respectively. TCL541 is priced at $2.74 and $5.00, respectively, for industrial and military orders in 100-piece orders.

Reader Service Number 11

Modem capable of voice and data switching

The POPCOM Model X100, a modem for users of personal computers and microcomputers, is available from Prentice Corporation.

The Model X100 is capable of true voice and data switching at the user's workstation, eliminating the need for separate lines for the telephone and personal computer. Dialing multiple calls to switch between voice and data is also eliminated. The voice and data handling feature also includes complete call progress monitoring and detection for dial tones, busy signals, remote ringing, voice, data, and even line current disconnect. Fully compatible with 103, 113, and 212A dial-up modems, the Model X100 connects to all standard single and multiline phone equipment.

POPCOM can be installed on the wall, desk, or floor. It is adjustment-free in operation (no switches or indicators) and provides answering for all voice and data calls, compatibility with currently available communications software packages, and the ability to adapt to a variety of RS-232 interface cables.

The Model X100 costs $475.

Reader Service Number 12

Artificial intelligence computer introduced for office application

Lisp Machine Inc. has introduced the Lambda/S, a compact, lower-priced version of its Lambda, an advanced LISP-based computer intended for artificial intelligence research and applications.

Equipped with the same 32-bit LISP processor as the full-sized Lambda, the Lambda/S proves similar computing power in a smaller package. It is supplied with a 169-megabyte Winchester disk drive and a 12-slot card cage.

The Lambda/S can be used as a stand-alone machine, and can also serve as a workstation on a local area network using a rack-mount Lambda or other host computer as a file-server. LISP software supplied with the machine can confer LISP-like file attributes on non-LISP file-servers. The Lambda/S is priced at $66,500.

With the introduction of the Lambda/S, LMI offers its LISP machines in three basic configurations. In addition to the compact Lambda/S, the firm's 32-bit LISP processor, which furnishes a 128M byte virtual address space, is available in a full-size rack configuration with either a 169M byte Winchester disk, priced at $69,900, or a 470M byte Winchester disk, priced at $76,500. The rack configuration of the Lambda offers a 21-slot card cage. An optional 68010-based UNIX processor is available.

The Lambda can be configured with LISP and 68010-based UNIX coprocessors in a high-speed, multiprocessor bus. This configuration enables the two processors to execute concurrently, communicating with each other through LMI's extended STREAMS interface. A dual-processor Lambda offers users the ability to add intelligence to an existing traditional software package operating in UNIX by placing it under the supervision of an evolving LISP program.

Lambda's virtual control store with microcompiler allows the machine's architecture to be rapidly and easily conformed to the requirements of a specific application. The LMI Lambda also features an integral Multibus, which allows system engineers a wide choice of lower-cost third party peripherals for configuring the system and an optional Ethernet-II interface for communication with other computers.

Reader Service Number 13



CAE tool simplifies simulation tasks for engineers

The MegaLogician Physical Modeling Extension, a CAE tool that performs simulation modeling of complex logic blocks, has been introduced by Daisy Systems.

PMX is a system design tool that integrates physical hardware subsystems and complex VLSI components, including complex VLSI microprocessors. When the component is plugged into a socket, the simulation information is directly acquired by extracting the behavior from the physical device itself. This virtually eliminates the breadboarding step previously necessary when complex components were utilized in a design.

PMX is an option to the MegaLogician, a CAE workstation with a simulation capacity of one million gates and a performance level of 100,000 evaluations per second. PMX supports all commercial VLSI components, customer-designed PC boards and proprietary semicustom and full custom logic. Included in the commercial component category are the microprocessors and peripheral component families of Intel, Motorola, National, AMD, and Texas Instruments.

PMX is priced from $15,000.

Reader Service Number 14

Simulation language designed for IBM PC

A software package that can be used to simulate complex business and engineering systems, GPSS/PC, has been introduced by Minuteman Software. GPSS/PC is an implementation of GPSS, the General Purpose Simulation System language.

GPSS/PC will be available for the IBM PC in May at a one-time license fee of $900.

Reader Service Number 15

Solid modeling is transferable from IBM PC to videotape

Cubicomp has introduced an add-on circuit board to allow videotaping of images created with the company's CS-5 solid modeling design system for the IBM PC. The new product, Video-5, increases the way solid modeling on microcomputers can be used in animation, advertising, engineering, architecture, CAD, and education.

Video-5 synchronizes the display generator of the Cubicomp CS-5 system with an external video signal so that, with the addition of a sync generator and color encoder, high quality images can be recorded in all standard video formats. CS-5 works with an IBM PC to allow users to create, display, manipulate, and store three-dimensional, full-color, shaded-surface solid models.

CS-5 allows up to 4096 colors to be used simultaneously to color and shade the surfaces of images so they have realistic depth and texture. Users can also create videotapes for use as portable demonstration tools. Tapes showing images in the progressive stages of design can be produced quickly.



Video-5 allows users to record images in either NTSC or PAL standard video formats. Images can be recorded on all videotape formats, including VHS and Beta home videotape as well as Umatic and Type C professional tape.

Video-5 is factory-installed in the display generating hardware of the CS-5 system. Its role is to link the display generator and the video recorder to a color encoder and color sync generator, two other standard pieces of equipment required to videotape computer graphics images. In essence, Video-5 locks the timing of the signal transmitted from the display generating hardware so that it can be properly received by the video recorder.

Priced at $900, Video-5 is available through Cubicomp. The CS-5 solid modeling system, consisting of display generating hardware and solid modeling software, is priced at $9,700.

Reader Service Number 16

New A/D converters feature low power consumption

Motorola announces availability of the MC145040/MC145041 analog-to-digital converters with serial interface. These 8-bit devices are compatible with SPI, Microwire, and similar interfaces.

The MC145040 features the following: operating supply voltage range of 4V to 6V for operation in NMOS as well as CMOS systems; operating temperature range of $-55°C$ to $+125°C$; and conversion rate of 16 microseconds for the standard part. The MC145041 also offers an on-chip oscillator that allows integration with low-cost MCUs that do not output a clock signal. The inputs are TTL-compatible and can be driven with CMSO parts as well. The 20-pin DIP packages provide for 11 analog channels. These devices will also be offered in plastic quad packages. Priced in 100-up quantities, the MC145040 is $2.89, and the MC145041 is $2.99.

Reader Service Number 17

The Product Summary is a review of selected product announcements received at *IEEE Micro*'s offices during the last two months. For information on any of these products, circle the appropriate Reader Service Number on the card at the back of the magazine.

IEEE MICRO'S PRODUCT SUMMARY

Product Summary Editor: Victor P. Nelson

MANUFACTURER & MODEL	FUNCTION	COMMENTS	R.S. NO.
Hewlett-Packard, ThinkJet	Printer	Ink-Jet printer interfaces with HP 150 (as well as IBM, Apple, and Compaq micros), operating at 150 cps. Combines printhead and ink reservoir in a single disposable unit with an average life of 500 pages. $495.	25
Digital Microsystems, Hinet PC Adapter	LAN adapter	One-slot board adds 64K RAM, a Z-80 processor, and an RS-232 port to the IBM PC, permitting compatibility with Hinet members using either CP/M, CP/M-86, or MS-DOS. $495.	26
Litton Systems, RS-232 Trackball Interface	RS-232 adapter	Permits compatibility with any device that has an RS-232 dataport (such as medical electronics components, training simulators, or arcade game devices). Self-contained within the TBSII package, unit eliminates the need for interface logic between trackball and computer. Price not stated.	27
RAD Computers, MME	Modem eliminator	Replaces two synchronous modems normally required to direct-connect two microcomputers. Unit requires no AC supply connection and operates at all standard rates from 1200 to 19,200 bps, with a range of 100 meters at 9600 bps. $150.	28
Interphase, ESDI 3191	Disk controller	Device supports Enhanced Small Device Interface (ESDI) for 5¼-inch Winchester disks. Compatible with all microcomputers based on the Intel multibus, the unit features trimode operations (buffered, direct, cached), disk data rates to 20 Mb/s, and self-diagnostic reporting. Single unit, $2250; under $1000 in quantity.	29
PC Technologies, Xtender	Cluster controller	Upgrades PC-XT (or IBM PC with hard disk) to serve as host for up to five independent terminals, creating a clustered processing environment. Offered with 512K or 1024K bytes of memory, board operates with a wide variety of ASCII terminals at speeds to 19,200 bps. Price not stated.	30
Eventide, WKBP-16	Memory board	Device requires no interface to fit Hewlett-Packard 9816, 9826, and 9836 computers. Memory quadruples density, giving the 9816 its first megabyte of RAM without using an expansion interface. Price not stated.	31
Intercontinental Micro, CPZ-4800X	S-100 bus SBC	Board supervises up to 16 slave processors and 4000 users in an Intercontinental ARC-100 network. Operating at clock rates of 4 MHz or 6 MHz, the unit's memory management and direct memory access provide data transfer rates 300 percent faster than Z-80 I/O block move rates. Including RS-232 communications and floppy controller personality boards, $982.	32

MANUFACTURER & MODEL	FUNCTION	COMMENTS	R.S. NO.
Qua Tech, 8-bit and 12-bit A/D Systems	A/D converter boards	Up to three modularized converters work in conjunction with a parallel expansion board to provide custom configurations to the user. Package includes Labstar software, which contains drivers for each module. Applications can be written in Basic or other assembly languages. 8-bit A/D system, $390; 12-bit A/D system, $690.	33
Qua Tech, IEEE 488 GPIB	Multifunction board for IBM	Board allows the IBM PC to emulate GPIB system controller, connecting up to 15 other talking, listening, or controlling devices for each module. Features programmable interval times and a parallel port that accepts modules such as A/D and D/A converters, reed relay and stepper motor controller. GPIB controller, $395; software support, $100; modules from $75 to $495.	34
Intel, iSBC 186/03, 186/78 and 86/35	Single-board computers	Based on Intel iAPX 186 and 86, computers provide complete system capabilities: CPU, operating system functions, peripheral interfaces, memory, and software. Company says new 6¾-inch boards replace several conventional boards and are cheaper, faster, and consume less power. All function under Intel's iRMX 86 operating system. iSBC 186, $1650; 186/78, $3000 (engineering samples); 86/35, $3495.	35
Cado Systems, Tiger ATS 16	Host microcomputer	Built around the Intel 80186, system supports as many as 16 terminals, printers, peripherals, or communications devices. With 512K RAM, a 10-Mb hard disk, and 1.2 Mb of diskette storage, the ATS 16 can simultaneously run accounting, word processing, financial planning, and message-processing applications. $12,895.	36
Romar Computer Systems, Romar IIX	Apple-compatible microcomputer	Unit features both Apple DOS and CP/M operating systems, a 6502-based CPU with ROM expandable to 192K, two slimline floppies, a Z-80 card, and eight expansion slots. Detachable keyboard has 87 keys, including numeric pad and function keys. $695.	37
Scientific Micro Systems, SMS 1000 Model 40	DEC-compatible system	Available in several configurations, all of which run application software for DEC LSI-11/23 or LSI-11/73 CPUs without modification. Comes in upright floor model or rack-mountable package with one or two 5¼-inch Winchesters (12 to 70 Mb each). Choice of one 8-inch floppy or two 5¼-inch, double-sided, RX50-compatible disk drives. Add-on cartridge tape subsystem is also available. From $5800.	38
Materials Development Corporation, RM-1600	IBM PC-compatible microcomputer	System mounts in standard 19-inch racks and features 9-inch screen, four expansion slots, and two IBM PC standard disk drives. Built-in graphics capability has a resolution of 640 × 325, over 50 percent higher than IBM's. $3995; with one floppy disk and one 10-Mb hard disk, $6495.	39
Wyse Technology, WY-50 with WyseWord	Dedicated word processing workstation	System integrates a firmware version of Wordstar with a 14-inch, nonglare green screen that tilts and rotates. Features user-selectable 80/132 column width and 32 preprogrammed functions on 16 keys. $795.	40
Microsoft, Microsoft Project	Project management package	Spreadsheet-style scheduler for the IBM PC provides individual resource and cost tables, graphic displays of schedules and resource utilization, and extensive reporting capability. Program offers instant "what-if" feature; built-in facility generates and prints predefined reports for analysis or presentation. For IBM and "strict" compatibles only; requires 128K and one DSDD drive. $250.	41

For more information, circle the appropriate RS No. on the Reader Service Card at the back of the magazine.

MANUFACTURER & MODEL	FUNCTION	COMMENTS	R.S. NO.
Micropro, Portable Wordstar, Calc, and Scheduler	Lap computer programs	Designed for the Epson PX-8, programs have both user file compatibility with their parent versions and are available in English, French, and German. Portable Wordstar retains most features of its namesake. Portable Calc permits independent setting of global, column, and cell formats, and variable column widths. Portable Scheduler combines the functions of an appointment book and travel alarm clock, storing many appointments at half-hour intervals. Price not stated.	42
Computer Colorworks, Flying Colors	Easy color graphics	Designed for use with Commodore 64 and a standard joystick, package features paint capabilities, thick/thin lines, and various sizes of automatic circles and boxes. Drawing speed is adjustable, and a grid feature permits easy alignment of figures. Pictures can be stored on disk and converted to slides. Also available for the Apple IIe and II+. $39.95.	43
Lytron Systems, LOIS	Umbrella environment for software	Package lets users integrate their favorite off-the-shelf software into a seamless, menu-driven shell environment allowing data transfer between applications on networks or through links to mainframes. LOIS supports Microsoft's announced windowing software, and versions are available on MS-DOS and PC-DOS. Introductory price, $295.	44
Lytron Systems, Office Organizer Tools	Office application software	Tools can operate as stand-alone applications or as components of LOIS. Scratchpad ($60) creates, distributes, and files notes and memoranda; Calendar ($135) schedules appointments and organizes memos; File Cabinet ($100) files and retrieves information in a variety of user-friendly methods; Calculator ($75) performs/displays manual or programmable calculations and supports scientific notation, memory manipulation, and scientific and financial modes. Total package, $495.	45
Radio Shack, RM/Fortran 77	Fortran compiler	Allows users to write and execute mainframe-level Fortran applications on the TRS-80 Model 16B microcomputer. Incorporates popular extensions such as Hollerith and dexadecimal constants, an "include" statement, and symbols of up to 31 characters. $699.	46
Supersoft, Diagnostics II	Software maintenance program	Now available for MS-DOS and PC-DOS, package checks all components of computer systems to verify correct operation. Comes as a stand-alone program for end users or as a customized package for OEMs. Tests five main areas of computer's system (CPU, memory, disk drives, terminal, and printer), isolating problems to the chip level. Price not stated.	47
Ashton-Tate, Framework	Integrated Software	Package combines word processing, spreadsheet, business graphics, data management, forms processing, and "outline operator" in a window-like environment. Compatible with dBase II, also provides access to Lotus 1-2-3 and other ASCII files. Runs PC-DOS software through a DOS access facility. $695.	48
Quickview, Viewdex	"Index card" filing system	Filing system for names, addresses, and phone numbers runs on Epson HX-20. Will also be introduced for NEC 8200 and Radio Shack TRS-80 Model 100. Features pop-up menus for beginning users; provides search, edit, and omit commands. Price not stated.	49

For more information, circle the appropriate RS No. on the Reader Service Card at the back of the magazine.

ACCESS
recent books and articles on microcomputing

by Peter R. Rony
Dept. of Chemical Engineering
Virginia Polytechnic Institute
and State University
Blacksburg, VA 24061

Articles

Business Computer Systems

Vol. 2, No. 11, Dec. 1983:

H. Swartz, "Parents Should Be Good Medicine for Software," pp. 29-30.

E. Horwitt, "Bringing Order to Project Jumble," pp. 60-71.

E. Benoit, "Business Takes Another Look at Lisa," pp. 74-80.

A. E. Smith, "Electronic Mail Goes First Class," pp. 84-91.

E. Horwitt, "More Bytes for the Buck," pp. 111-128 [Note: Includes table of Winchester disk drives].

D. Pryor and A. Smith, "Power Line Protection," pp. 131-41 [Note: Includes extensive buyer's guide to power line protection].

J. Schlosser, "TK!Solver Offers Solutions to Complex Business Problems," pp. 151-157.

Vol. 3, No. 1, Jan. 1984:

H. Swartz, "A Watershed Case for Copyright," pp. 25-26.

E. Ferrarini, "Using Data Bases to Search for Software," pp. 35-39.

T. Johnson, "The Year Ahead," pp. 42-73 [Note: Opinions of experts, users, and dealers on trends, coups, products, and so forth].

E. Benoit, "Annual Report: Floppy Disk Drives," pp. 111-121 [Note: Includes extensive table of floppy disk drives].

Byte

Vol. 9, No. 1, Jan. 1984:

B. Robertson, "California Hardware," pp. 52-54.

Special section—"1984 and Beyond." Articles include "Reason and the Software Bus," "A General-Purpose Robot-Control Language," "1984, the Year of the 32-Bit Microprocessor," "Memory Cards: A New Concept in Personal Computing," "Computer-Aided Design" [Note: Includes comparison of primitive features of CAD packages and a glossary of CAD terms], "Speech Recognition: An Idea Whose Time Is Coming," "Using Natural-Language Systems on Personal Computers," "Portables—1984 and Beyond: Idea-Processing Software and Portable Computers," and "Beyond the Application Program: A Different Approach to Integrated Software," pp. 100-262.

K. Skier, "The Zenith Z100," pp. 268-278.

S. Barry and R. Jacobson, "The TRS-80 Model 16B With Xenix," pp. 288-320.

M. Haas, "Naturallink to Dow Jones News/Retrieval," pp. 324-334.

L. Wheeler, "Bubbles on the S-100 Bus: Part 1—The Hardware," pp. 362-380.

J. T. Maxwell III and S. M. Ornstein, "Mockingbird: A Composer's Amanuensis," pp. 384-401.

E. M. Carter and A. B. Bonds, "The VU68K Single-Board Computer," pp. 403-416.

J. Bass, "Translating the SAS Language Into Basic," pp. 417-434.

D. K. Broadwell, "Real-Time Clocks and PC-DOS 2.0," pp. 442-450.

Vol. 9, No. 2, Feb. 1984:

G. Williams, "The Apple Macintosh Computer," pp. 30-54.

"An Interview: The Macintosh Design Team," pp. 58-80.

S. Ciarcia, "Build the Circuit Cellar Term-Mite ST Smart Terminal: Part 2—Programming and Use," pp. 88-109.

Special section—"Benchmarks and Performance Evaluation." Articles include "Don't Bench Me In," "Beyond MIPS: Performance Is NOT Quality," "Software Performance Evaluation," "The Art of Benchmarking Printers," "Benchmarking Fortran Compilers," "Benchmark Confessions," "The Word Processing Maze," "Evaluating Word Processing Programs," pp. 158-246.

Books

H. Anbarlian, *An Introduction to Visicalc Spreadsheeting for the TRS-80 Model III,* McGraw-Hill Book Co., 1983, 434 pp., $49.95 (includes disk).

H. Anbarlian, *An Introduction to Visicalc Spreadsheeting for the TRS-80 Model II and Model 16,* McGraw-Hill Book Co., 1983, 350 pp., $49.95 (includes disk).

M. Banahan and A. Rutter, *The Unix Book,* John Wiley & Sons, Inc., 1983, 218 pp., $16.95.

M. Brady et al., eds., *Robot Motion: Planning and Control,* MIT Press, 1983, 585 pp., $37.50.

S. Crop and D. Mosher, *Portable Computers,* Sybex Inc., 1984, 103 pp., $7.95.

T. A. Dwyer and M. Critchfield, *Pocket Guide to CP/M,* Addison-Wesley Publishing Co., 1983, 50 pp., $7.25.

L. Finkel and J. Brown, *TRS-80 Data-File Programming,* John Wiley & Sons, Inc., 1983, 306 pp., $14.95.

R. H. Flast, *54 Supercalc Models,* Osborne/McGraw-Hill, 1983, 244 pp., $15.95.

O. Kellogg, *The Radio Shack Notebook Computer,* Sybex Inc., 1984, 118 pp., $8.95.

T. King and B. Knight, *Programming the M68000,* Addison-Wesley Publishing Co., 1983, 154 pp., $12.95.

F. T. Leighton, *Complexity Issues in VLSI,* MIT Press, 1983, 139 pp., $19.95.

T. G. Lewis, *Pascal for the IBM Personal Computer,* Addison-Wesley Publishing Co., 1983, 406 pp., $15.95.

D. Müller, *Dictionary of Microprocessor Systems,* Elsevier Science Publishing Co., Inc., 1984, $67.50.

L. Poole, *Apple II User's Guide,* Osborne/McGraw-Hill, 1983, 482 pp., $17.95.

J. Sachs, *Your IBM PC Made Easy,* Osborne/McGraw-Hill, 1984, 438 pp., $12.95.

P. A. Sand, *Advanced Pascal Programming Techniques,* Osborne/McGraw-Hill, 1984, 370 pp., $19.95.

M. Waite, D. Martin, and S. Prata, *Unix Primer Plus,* Howard W. Sams & Co., Inc., 1983, 414 pp., $19.95.

M. Waite and C. Morgan, *Graphics Primer for the IBM PC,* Osborne/McGraw-Hill, 1983, 440 pp., $21.95.

R. Moore, "ProDOS" [software review], pp. 252-262.

T. R. Clune, "The IBM CS-9000 Lab Computer," pp. 278-290.

C. Weger, "The Rixon R212A Intelligent Modem," pp. 292-300.

R. Jones, "IBM/Apple Communication," pp. 331-340.

J. D. Blagg, "A Low-Cost, Low-Write-Voltage EEPROM," pp. 343-344.

K. Christian, "Inside a Compiler: Notes on Optimization and Code Generation," pp. 349-366.

J. E. Roskos, "Writing Device Drivers for MS-DOS 2.0 Using Tandon TM100-4 Drives," pp. 370-380.

R. Sussman and T. Sussman, "Five Original Graphics," pp. 388-391.

L. Wheeler, "Bubbles on the S-100 Bus: Part 2—The Software," pp. 395-414.

D. F. Hinnant and M. B. Smith, "Bullet-Proof Pascal Input," pp. 428-434.

J. L. Star, "Favorite Benchmarks," p. 436.

Computer

Vol. 17, No. 1, Jan. 1984:

J. Grinberg, G. R. Nudd, and R. D. Etchells, "A Cellular VLSI Architecture," pp. 69-81.

J. W. Balde, "IEEE Computer Packaging Committee Spring Packaging Workshop," pp. 83-86.

Vol. 17, No. 2, Feb. 1984:

H. U. Steusloff, "Advanced Real-Time Languages for Distributed Industrial Process Control," pp. 37-46.

Datamation

Vol. 30, No. 1, Jan. 1984:

J. Raloff, "Keeping Pace," pp. 38-46.

G. D. Brown and D. H. Sefton, "The Micro vs. the Applications Logjam," pp. 96-104.

M. Tyler, "Touch Screens: Big Deal or No Deal?" pp. 146-154.

Vol. 30, No. 2, Feb. 1984:

Special issue on IBM. Articles include "The Song Remains the Same," "Battle of the Baby Blue Boxes," "Big Blue's Big Bucks," "An End to Handholding," "With a Little Help From Some Friends," "The Legend of the Jolly Blue Giant," "Mainframe Maneuvers," "Software Strategies," and "Assessing PROFS," pp. 104-192.

Dr. Dobb's Journal

Vol. 9, No. 1, Jan. 1984:

N. C. Shammas, "NBasic: A Structured Preprocessor for MBasic," pp. 24-34.

E. Mitchell, "A Simple Window Package," pp. 36-43.

D. Cornell, "Forth to PC-DOS Interface," pp. 44-59.

D. Daetwyler, "Sorted Diskette Directory Listing for the IBM PC," pp. 60-74.

Vol. 9, No. 2, Feb. 1984:

K. Coye and A. Grossman, "Micro to Mainframe Connection," pp. 20-24.

L. Brooks, "Communications Protocols: Theory and Practice," pp. 26-29.

J. Fleming, "Virtual Personal Computer," pp. 32-59.

R. S. Broughton, "Basic Language Telecommunications Programming," pp. 64-69.

EDN

Vol. 28, No. 24, Nov. 24, 1983:

W. Twaddell, "Standard-Cell Design Inroads Spur Specialized Gate-Array Growth," pp. 49-58.

R. E. Peterson, Jr., "Flat-Panel Displays," pp. 102-127.

A. Rappaport, "Hands-On Timing Verification Validates IC Design Techniques," pp. 147-162.

A. J. Hennigan and E. Zanelli, "High-Power, Dual-Bridge ICs Ease Stepper-Motor-Drive Design," pp. 167-177.

V. J. Coli, C. Hastings, and S. Rajpal, "Bipolar Arithmetic Chip Speeds 68000's Math Throughput," pp. 179-193.

B. Collis, "Macros Speed 8080, Z80 Multiplication," p. 225.

M. W. Johnson, "Absolute-Value Code Executes Quickly," p. 228.

Vol. 28, No. 25, Dec. 8, 1983:

B. Travis, "High-Performance Single-Chip μCs Provide Powerful Options," pp. 52-73.

G. Legg, "Minicomputer-Hosted Compilers Ease μP Software Development," pp. 270-276.

R. Comerford, "High-Level Graphics Tools Hasten Data Interpretation," pp. 40-48.

T. Ormund, "Latest Keyboard Improvements Stress Ergonomic Considerations," pp. 106-116.

C. H. Small, "Enhanced Signature Analyzers Ease μP-System Diagnosis," pp. 168-178.

Vol. 29, No. 1, Jan. 12, 1984:

A. Rappaport, "Semicustom-IC Users' Needs Determine Technology Development," pp. 75-86.

J. McDermott, "Digital Motion Control," pp. 124-152.

J. E. Hemenway, "Powerful VME Bus Features Ease High-Level μC Applications," pp. 158-168.

"1983 Technical Article Database Index," pp. 177-250 [Note: An index of articles that appeared in *EDN, Electronic Design, Electronics, Electronic Products,* and *Computer Design*].

D. Garde, "Single-Port Multiplier Reduces DSP System Costs," pp. 277-286.

Electronics

Vol. 57, No. 1, Jan. 12, 1984:

S. Joshi and V. Iyer, "Protocols and Network Control Chips: A Symbiotic Relationship," pp. 157-163.

S. J. Frank, "Tightly Coupled Multiprocessor System Speeds Memory Access Times," pp. 164-169.

Vol. 57, No. 2, Jan. 26, 1984:

E. L. Keller, "Automotive Electronics Shift Into Overdrive," pp. 101-112.

J. Munday, "Three-Chip Set Offers Universal Cost-Effective SLIC," pp. 116-119.

N. R. Vemula, "Single IC Can Perform Speech Recognition and Synthesis," pp. 120-122.

Vol. 57, No. 3, Feb. 9, 1984:

E. Newman, "Personal Computers Invade Power Arena," pp. 106-108.

M. A. Harris, "Computer Makers Bet on Standards," pp. 110-111.

P. D. MacWilliams, P. I. Woolchow, and R. H. Zasloff, "Microcomputer Serial Bus Yields Distributed Multispeed Control," pp. 125-134.

D. Way, "Front-End Processors Smooth Local Network-Computer Integration," pp. 135-139.

R. Wirfs-Brock and W. Dodge, "Highly-Integrated Processor Eases Design of Low-Cost Display Station," pp. 140-143.

Elektor

Vol. 9, No. 12, Dec. 1983:

"Novram: Data Storage Without Batteries," pp. 58-61.

"Bus Extension," pp. 62-63.

Vol. 10, No. 1, Jan. 1984:

"Z80 EPROM Programmer," pp. 60-61.

Forth Dimensions

Vol. 5, No. 4, Nov./Dec. 1983:

H. Laxen, "Multi-Tasking: Part I," pp. 26-28.

Hewlett-Packard Journal

Vol. 35, No. 1, Jan. 1984:

L. M. Koehler and T. C. Mackey, "Speech Output for HP Series 80 Personal Computers," pp. 29-36.

IEEE Spectrum

Vol. 21, No. 1, Jan. 1984:

"Best Bits: Applications of Microprocessors—Watching for Criminals, Micro Beats Chess Master, and Running Motors Efficiently," p. 26.

Special section—"Technology '84." Articles include "The Penetrating Technologies: An Analysis by the Experts," "Personal Computers," "Software," "Microprocessors," "Communications," "Solid State," "In-

strumentation," "Industrial Electronics," "Power and Energy," "Consumer Electronics," "Rail Transportation," "Aerospace and Military," "Medical Electronics," and "Education," pp. 34-98.

Vol. 21, No. 2, Feb. 1984:

H. Morkoc and P. M. Solomon, "The HEMT: A Superfast Transistor," pp. 28-35.

S. B. Weinstein, "Smart Credit Cards: The Answer to Cashless Shopping," pp. 43-49.

Interface Age

Vol. 9, No. 2, Feb. 1984:

C. Heintz, "Solving the Data Base Puzzle," pp. 58-69.

Microprocessing and Microprogramming (The Euromicro Journal)

Vol. 12, Nos. 3-4, Oct./Nov. 1984:

Special issue containing "short notes" from Euromicro '83. Articles include "A High-Level Language Architecture: Bit-Slice-Based Processor and Associated System Software," "Modular, Object-Oriented Microcomputer Architecture," "Using the IAPX-432 System as a Support for Chill Parallel Constructs," "Small Local Area Network With a Collision-Free Technique for Connecting Heterogeneous Microcomputer Systems," "A Multi-Microprocessor-Based Intelligent Graphics Terminal," "Mirtex: A Real-Time, Multi-Tasking Executive for Microprocessors," "The Modhel Microcomputer for RICS Study," "Microprocessor-Controlled Data Acquisition System for Radio Propagation Measurements," "Microsimplex: A Didactic Microcomputer Specially Designed for Computer Education," "CAI System for Basic," "Bandwidth Reduction for Speech Transmissions: A Microprocessor-Based Simulator," and "DORMS—A Design Tool for Multiple Microprocessor Systems," pp. 143-244.

Vol. 12, No. 5, Dec. 1983:

G. Bolch et al., "A Multiprocessor System for Simulating Data Transmission Systems (Mupsi)," pp. 267-277.

G. Fazio and A. Salsano, "A Fault-Tolerant Microcomputer With Fail-Save Outputs," pp. 279-284.

B. Kaminska and J. Zabrodski, "The Concept of Proper Operation Region in Digital Integrated Circuit Testing, The 8080A Microprocessor as an Example," pp. 285-290.

A. Rijllart, "An MC68000 Stand-Alone CAMAC Microprocessor System," pp. 291-297.

Microprocessors and Microsystems

Vol. 7, No. 9, Nov. 1983:

R. Barron, A. A. Akinola, and D. Reid, "Use of Microcomputers in Monitoring Instruments, Data Acquisition, Analysis, and Control," pp. 419-428.

S. P. Lee, J. T. Lue, and H. Yu, "Software Development for Communication Between Two Microcomputers," pp. 429-433.

P. Twomey and C. O'Sullivan, "Mobile Digital Data Link for Field Work Applications," pp. 435-438.

B. Srinivasan, "Methodology for the Evaluation and Selection of Microprocessors for Specific Applications," pp. 439-443.

M. F. Smith, "Multichannel Software Event Counter for the MC6809," pp. 444-445.

Vol. 7, No. 10, Dec. 1983:

J. So, "TMS 320—A Step Forward in Digital Signal Processing," pp. 451-460.

S. Y. Kung and J. Annevelink, "VLSI Design for Massively Parallel Signal Processors," pp. 461-468.

Y. Suzuki et al., "Advanced C^2MOS Speech Synthesizers," pp. 469-474.

P. Challener, "FAD—Flexibility in Digital Signal Processing," pp. 475-481.

A. G. Corry et al., "Image Processing With VLSI," pp. 482-486.

Mini-Micro Systems

Vol. 16, No. 14, Dec. 1983:

Special issue on technology includes articles on "Computer Architecture," "Microprocessors," "Application Processors," "Database Management Systems," "Software Development," "Office Automation," and "Artificial Intelligence," pp. 145-233.

Vol. 17, No. 1, Jan. 1984:

Special issue on desktop printers:

E. S. Foster, "Printers and Plotters Vie for Color Hard-Copy Market," pp. 100-104.

S. Glazer, "Competition Heats Up in Growing Fault-Tolerant Computer Market," pp. 111-126.

S. Glazer, "Business Graphics for Unix Covers a Small Canvas," pp. 135-144.

M. Stiefel, "Product Profile: Correspondence-Quality Serial Printers," pp. 152-190.

E. Webster, " 'New Technology' Printers Challenge Daisy Wheels," pp. 197-204.

R. A. Williams, "Matrix Printer Combines Letter-Quality Text, Color Graphics," pp. 211-218.

C. C. Yang, "Ink-Jet Printer Handles Text, Graphics," pp. 223-228.

G. Legg and R. Friedman, "Microcomputer Operating Systems Compete in Portability and Performance," pp. 233-261.

Vol. 17, No. 2, Feb. 1984:

S. J. Shaw, "Harris Poll Shows Public Is High on Computers," p. 55.

J. Tarter, "Micro Software Industry Faces Clogged Distribution Channels," pp. 131-140.

D. R. Simpson, "Product Profile: Half-Height 5 1/4-inch Disk Drives Will Dominate Low-End Storage Market," pp. 176-202.

D. Simpson, "Streaming-Tape Cartridge and Cassette Drives Extend Back-Up Abilities," pp. 225-232.

C. Czernek, "Understanding Head-Positioning Systems in Winchester Drives," pp. 237-248.

M. Simmons, "How to Select Jumper Options for Floppy Disk Drives," pp. 253-262.

J. Liu, "Half-Height 5 1/4-inch Winchesters Adopt Large-Drive Technology," pp. 267-272.

R. R. Friedman, "Word-Processing Software for Minicomputers Rivals Capability of Turnkey Systems," pp. 277-299.

PC World

Vol. 2, No. 2, Feb. 1984:

W. Claxton, "PC-TV," pp. 48-59.

K. Seger and F. E. Davis, "IBM's Home Run," pp. 96-101.

K. Seger and A. Mello, "Charting the PC Jr. Software Course," pp. 114-118.

L. Jordan, "A Host of Hosts," pp. 132-143.

C. Huizenga and C. Barnaby, "But Is It Really Fortran?" pp. 172-179.

S. Manes, "Making a Date," pp. 212-218.

G. I. Ouchi, "Lotus in the Lab," pp. 222-229.

Vol. 2, No. 3, Mar. 1984:

D. Benchley, "Occam's Edge," pp. 60-65.

J. Fox, "Big Ideas From Smalltalk," pp. 72-75.

J. Martin and S. R. Schwartz, "Good Impressions," pp. 84-92.

E. Weiss, "/X Marks the Macro," pp. 164-172.

B. Mann, "Changing Keys," pp. 179-181.

J. K. Mickelsen, "Really Random," pp. 188-193.

M. A. Covington, "Taming the Pascal Compiler," pp. 202-208.

Systems and Software

Vol. 2, No. 12, Dec. 1983:

R. Bernhard, "Data Terminals," pp. 104-112.

R. Bird, "Graphics Processor Lowers System Cost," pp. 117-122.

E. A. Snyder, "File Processor Speeds I/O," pp. 125-128.

M. W. Eisele, "Bubble Memories Are Back," pp. 131-134.

"Network Software," "Liason: Opening the Networking Applications Market," and "CP/Net: CP/M Shaped to a Local Network," pp. 137-148.

W. Rauch-Hindin, "Artificial Intelligence: Part 1," pp. 150-177.

Vol. 3, No. 1, Jan. 1984:

J. McLeod, "Will IBM's Micro Conquests Continue With Popcorn?" pp. 59-62.

J. Braun, "Menus Power Up CP/M for Supermicro," pp. 141-144.

W. Rauch-Hindin, "Natural Language: Part 2," pp. 187-230.

Vol. 3, No. 2, Feb. 1984:

W. Crutchley, "Developing Graphics Displays Faster," pp. 105-110.

C. Holsapple, "Synergy Comes to Integrated Software," pp. 113-120.

I. Kalb, "A More Friendly Command Interpreter," pp. 123-126.

M. Chester, "Small Business Systems," pp. 128-142.

D. Nelbaur, M. Hurst, K. Powell, and D. Major, "Synchronizing Shared Files," pp. 145-152.

W. Cassell, "Turning the IBM XT Into a Multi-Terminal System," pp. 152-156.

J. McLeod, "While Media Size Contracts, Capacity Continues to Grow," pp. 159-166.

Addresses of publishers

Addison-Wesley Publishing Co.
Jacob Way
Reading, MA 01867
(617) 944-3700

Business Computer Systems
Cahners Publishing Co.
270 St. Paul St.
Denver, CO 80206
(303) 388-4511
(Publisher does not sell back issues or single copies of articles. Subscriptions are free to qualified readers.)

Byte
Byte Publications Inc.
70 Main St.
Peterborough, NH 03458
(603) 924-9281

Computer
IEEE Computer Society
10662 Los Vaqueros Cir.
Los Alamitos, CA 90720
(714) 821-8380

Datamation
Technical Publishing Co.
1301 S. Grove Ave.
Barrington, IL 60010
(312) 774-8115

Dr. Dobb's Journal
People's Computer Co.
Box E
1263 El Camino Real
Menlo Park, CA 94025
(415) 323-3111

EDN
Cahners Publishing Co.
270 St. Paul St.
Denver, CO 80206
(303) 388-4511

Electronics
McGraw-Hill Bldg.
1221 Ave. of the Americas
New York, NY 10020
(212) 997-1221

Elektor
Elektor Publishers Ltd.
6 Stour St.
Canterbury CT1 2XZ UK

Elsevier Science
Publishing Co., Inc.
52 Vanderbilt Ave.
New York, NY 10017
(212) 867-9040

Forth Dimensions
Forth Interest Group
PO Box 1105
San Carlos, CA 94070

Hewlett-Packard Journal
Hewlett-Packard Co.
3000 Hanover St.
Palo Alto, CA 94304
(415) 857-3853

IEEE Spectrum
IEEE Service Center
455 Hoes Lane
Piscataway, NJ 08854
(201) 981-0060, x133

Interface Age
McPheters, Wolfe & Jones
16704 Marquardt Ave.
Cerritos, CA 90701
(213) 926-9540

McGraw-Hill Book Co.
1221 Ave. of the Americas
New York, NY 10020
(212) 997-1221

Microprocessing and Microprogramming (The Euromicro Journal)
Elsevier/North-Holland, Inc.
52 Vanderbilt Ave.
New York, NY 10017
(212) 867-9040

or

North-Holland Publishing Co.
PO Box 211
1000 AE Amsterdam
The Netherlands

Microprocessors and Microsystems
Business Press International
205 E. 42nd St., Suite 1705
New York, NY 10017
(212) 867-2080

or

Butterworth Scientific Ltd.
Journals Division
PO Box 63
Westbury House
Bury St.
Guildford, Surrey GU2 5BH UK

Mini-Micro Systems
Cahners Publishing Co.
270 St. Paul St.
Denver, CO 80206
(303) 388-4511

MIT Press
28 Carleton St.
Cambridge, MA 02142
(617) 253-2884

Osborne/McGraw-Hill
630 Bancroft Way
Berkeley, CA 94710
(800) 227-2895,
in CA (415) 548-2805

PC World
PO Box 6700
80 New Bridge Rd.
Bergenfield, NJ 07621
(415) 861-3861
(800) 247-5470,
in IA (800) 532-1272

Howard W. Sams & Co., Inc.
4300 W. 62nd St.
PO Box 7092
Indianapolis, IN 46206
(800) 428-3696,
in IN (317) 298-5566

Sybex Inc.
2344 Sixth St.
Berkeley, CA 94710
(800) 227-2346,
in CA (415) 848-8233

Systems and Software
Hayden Publishing Co., Inc.
50 Essex St.
Rochelle Park, NJ 07662
(201) 843-0550

John Wiley & Sons, Inc.
One Wiley Dr.
Somerset, NJ 08873
(201) 469-4400

ADVERTISER/PRODUCT INDEX

ADVERTISERS—June 1984

Compcon Fall '84 Cover IV

IEEE Computer Society Press 32

Microprocessor Forum Cover II

Phoenix Software Associates Ltd. 3,5

FOR DISPLAY ADVERTISING INFORMATION CONTACT

Southern California and Mountain States: Richard C. Faust Company, 24050 Madison Street, Suite 100, Torrance, CA 90505; (213) 373-9604.
Northern California and Pacific Northwest: Don Farris Company, 616 Ramona Street, #20, Palo Alto, CA 94301; (415) 327-4100.
Jack Vance, P.O. Box 3205, Saratoga, CA 95070, (408) 741-0354
East Coast: Hart Associates, Inc., P.O. Box 339, 42 Lake Blvd., Matawan, NJ 07747; (201) 583-8500.
New England: Arpin Associates, P.O. Box 227, Weston, MA 02193; (617) 899-5613.
George Watts, III, 4 Conifer Dr., Wilbraham, MA 01095; (413) 596-4747.
Midwest: Jerry Berk, Special Market Media, 421 Florence Ave., Evanston, IL 60202; (312) 271-2001.
Southeast: Gerald F. Sweeney & Associates, P.O. Box 662, New York, NY 10113; (212) 242-3540
Southwest: The House Company, 3817 Richmond Avenue, Suite 110, Houston, TX 77027; (713) 622-2868.
Canada: Peter Drake & Associates, 17 Pine Crescent, Toronto, Ontario M4E 1L1, Canada(416) 690-2423.

Advertising Manager: Mike Koehler, *IEEE MICRO*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720, (714) 821-3240, 821-8380,

For production information, conference or classified advertising contact Don Farnsworth, *IEEE MICRO*, 10662 Los Vaqueros Circle, Los Alamitos, CA 90720, (714) 821-1140.

Moving?

PLEASE NOTIFY US 4 WEEKS IN ADVANCE

MAIL TO:
IEEE Service Center
445 Hoes Lane
Piscataway, NJ 08854

Name (Please Print)

New Address

City State/Country Zip

ATTACH LABEL HERE

- **This notice of address change will apply to all publications to which you subscribe.**
- **List new address above.**
- **If you have a question about your subscription, place label here and clip this form to your letter.**

PRODUCTS

	RS#	Page#
BOARDS		
Cluster controller	30	81
Converter	5,17,33	76,80,82
LAN adapter	26	81
Memory	31	81
Multifunction	34	82
Solid modeling	16	80
S-100 bus	32	81
CHIPS/COMPONENTS		
Micro enhancement	3	76
RS-232 adapter	27	81
CIRCUITS		
Data acquisition	11	79
CONFERENCES		
Compcon fall		Cover IV
Microprocessors		Cover II
DATA COMMUNICATIONS		
Data sentry modem	10	78
Modem emulator	28	81
Voice/data switching modem	12	79
I/O RELATED EQUIPMENT		
Mouse	9	78
Printer	25	81
MEMORY/STORAGE EQUIPMENT		
Disk controller	29	81
PUBLISHERS		
Computer Society publications		32
SOFTWARE		
CAE tool	14	80
Compiler	46	83
Emulation tools	4	76
Filing	49	83
Graphics	43	83
Integrated software	48	83
Lap computer	42	83
Logic simulation	1	75
Maintenance	47	83
Modular programming	21	5
Office applications	45	83
Program editing	20	3
Programming	6	77
Project management	41	82
Simulation	15	80
Umbrella environment	44	83
UNIX	7	77
SYSTEMS		
Artificial intelligence	13	79
DEC compatible	38	82
Microcomputer	2,36, 37,39	75,82, 82,82
Single-board computer	35	82
Voice recognition	8	77
Workstation	40	82

IEEE MICRO READER SERVICE

For further information on advertised products, new products, or literature, fill out one of these cards. Circle the number on the Reader Service Card that corresponds to the number at the bottom of the item in which you are interested. Affix stamp and mail.

Please print or type your name and address.

READER SERVICE CARD

IEEE MICRO

Void after December 15, 1984 **6/84**

Name _____

Company _____

Address _____

City _____ State _____ Zip _____

Title _____

Telephone number () _____

101 ☐ Please send me the IEEE-CS *1984 Publications Catalog.*

102 ☐ Please send me an IEEE fellows nomination form.

Articles I liked this issue:

A ☐ Software Driver for AM9511

B ☐ System Executive

C ☐ P959 Draft

D ☐ Reduced Instruction Set

E ☐ Microview

F ☐ Microstandards

G ☐ Microreview

H ☐ Microlaw

PRODUCTS PURCHASED OR SPECIFIED	FOR JOB	FOR HOBBY
Computer	☐	☐
Peripherals	☐	☐
Data communications	☐	☐
Memories, components	☐	☐
Software and services	☐	☐
Publications	☐	☐
Other	☐	☐

Send more information on numbered items:

1	5	9	13	17	21	25	29	33	37	41	45	49	53	57	61	65	69	73	77	81	85	89	93	97
2	6	10	14	18	22	26	30	34	38	42	46	50	54	58	62	66	70	74	78	82	86	90	94	98
3	7	11	15	19	23	27	31	35	39	43	47	51	55	59	63	67	71	75	79	83	87	91	95	99
4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80	84	88	92	96	100

Comments on articles and other editorial matter

I liked: _____

I disliked: _____

I would like: _____

For reader service, see other side

This PO box for reader service cards only.

PLACE STAMP HERE

IEEE MICRO

Reader Service Inquiries
Box 24168
Los Angeles, CA 90024
USA

READER SERVICE

OFESSIONAL ENDAR

1984

n Fall 84, Small Computer ation, September 16-20, Ar- ia. Contact Compcon Fall 84, Silver Spring, MD 20901; (301) K 7108250437 IEEECOMPSO.

984

utorial Week West 84, October Francisco, California. Contact L 72), Lawrence Livermore Lab- ox 806, Livermore, CA 94550;

oftware Engineering Standards ion Workshop, VLSI in Com- r 2-4, San Francisco, California. ard L. Tripp, Boeing Computer ox 24346, MS 9C-70, Seattle, WA 5-5390.

4, International Conference on er Design, VLSI in Computers, Port Chester, New York. Con- PO Box 639, Silver Spring, MD 589-8142; TWX 7108250437 O.

onference on Local Computer ks, October 8-10, Minneapolis, Contact Harvey Freeman, PO inneapolis, MN 55424.

International Conference on Advances in Microprocessor Peripherals, October 9-11, Lausanne, Switzerland. Contact Secretariat des Journeés d'Électronique et de Microtechnique, Departement d'Electricité – EPFL, Chenin de Bellerive 16, CH-1007 Lausanne, Switzerland; phone 41-21-471111; telex 26420.

1984 IEEE Ada Applications and Environments Conference, October 15-18, St. Paul, Minnesota. Contact Joe Urban, Computer Science Dept., University of Southwestern Louisiana, Lafayette, LA 70504; (318) 231-6304.

1984 International Test Conference, October 16-18, Philadelphia, Pennsylvania. Contact Doris Thomas, PO Box 371, Cedar Knolls, NJ 07927; (201) 267-7120.

Iecon 84, 1984 International Conference on Industrial Electronics, Control, and Instrumentation (IEEE et al.), October 22-26, Tokyo, Japan. Contact F. Harashima, Institute of Industrial Science, University of Tokyo, Roppongi, Minato-ku, Tokyo 106, Japan; phone 03-402-6231.

Workshop on Automotive Applications of Microprocessors, October 25-26, Detroit, Michigan. Contact John G. Neuman, RMB-392, Electrical Engineering Dept., General Motors Research Laboratories, Warren, MI 48090-9055.

Micro-17, 17th Annual Microprogramming Workshop (ACM Sigmicro, Euromicro), October 29-November 1, New Orleans, Louisiana. Contact Bruce Shriver, Computer Science Dept., University of Southwestern Louisiana, PO Box 44330, Lafayette, LA 70504; (318) 231-6606.

November 1984

Compsac 84, Eighth International Computer Software and Applications Conference, November 5-9, Chicago, Illinois. Contact Nick Marselos, AT&T Technology, 2600 Warrenville Rd., Lisle, IL 60532; (312) 260-4104.

FGCS 84, International Conference on Fifth Generation Computer Systems, November 6-9, Tokyo, Japan. Contact FGCS 84 Secretariat, Institute for New Generation Computer Technology, Mita Kokusai Bldg. 21F, 1-4-28 Mita, Minato-ku, Tokyo 108, Japan.

Third Annual Workshop on Computers and the Handicapped (ACM), November 8-9, Wichita, Kansas. Contact Elmer Hoyer, Wichita State University, Box 44, Wichita, KS 67208; (316) 689-3415.

1984 Workshop on VLSI Signal Processing Systems (IEEE et al.), November 12-14, Los Angeles, California. Contact Graham R. Nudd, Hughes Research Laboratories, 3011 Malibu Canyon Rd., Malibu, CA 90265.

December 1984

IEEE Workshop on Principles of Knowledge-Based Systems, December 3-4, Denver, Colorado. Contact John Roach, Dept. of Computer Science, 562 McBryd Hall, Virginia Polytechnic Institute and State University, Blacksburg, VA 24061; (703) 961-5368.

International Conference on Computers, Systems and Signal Processing, December 10-12, Bangalore, India. Contact I. G. Sarma, School of Automation, Indian Institute of Science, Bangalore 560 012 India; phone 34411; telex 0845 8349; or V. Ramachandran, Concordia Univ., 1455 De Maisonneuve Blvd. W., Montreal, PQ 43G 1M8, Canada; (514) 879-2826.

Mini and Microcomputers in Control, Filtering, and Signal Processing, December 10-12, Las Vegas, Nevada. Contact G. T. Catalano, Boeing Commercial Airplane Co., MS 69-79, PO Box 3707, Seattle, WA 98124; (206) 237-0709.

February 1985

32nd Annual IEEE International Solid-State Circuits Conference, February 13-15, New York, New York. Contact: Lewis Winner, 301 Almeria, Coral Gables, FL 33134; (305) 446-8193 or -8194.

Compcon, Spring 85, Technological Leverage: A Competitive Necessity, February 25-28, San Francisco, California. Contact Glen G. Langdon, Jr., IBM Dept. K54/282, 5600 Cottle Rd., San Jose, CA 95193; (408) 256-6454.

March 1985

Fourth Annual Phoenix Conference on Computers and Communications, March 20-22, Phoenix, Arizona. Contact IEEE PCCC-85, PO Box 37125 C, Phoenix, AZ 85069.

IEEE Infocom 85, March 25-28, Arlington, Virginia. Contact IEEE Infocom 85, PO Box 639, Silver Spring, MD 20901; (301) 589-8142; TWX 7108250437 IEEECOMPSO.

1985 Microprocessor Forum, March 31-April 4, Atlantic City, New Jersey. Contact N. R. Kornfield, Widener University, School of Engr., Chester, PA 19013; (215) 499-4055.

June 1985

FTCS-15, 15th International Symposium on Fault-Tolerant Computing, June 19-21, Ann Arbor, Michigan. Contact John F. Meyer, Dept. of Electrical Engr. and Comp. Sci., University of Michigan, Ann Arbor, MI 48109; (313) 763-0037; telex 810-223-6056.

DAC 85, 22nd Design Automation Conference (ACM), June 23-26, Las Vegas, Nevada. Contact Hillel Ofek, IBM (B2211951), PO Box 390, Poughkeepsie, NY 12602; (914) 485-7763.

August 1985

Eighth International Conference on Software Engineering (ACM), August 27-30, London, England. Contact Eighth ICSE, PO Box 639, Silver Spring, MD 20901; (301) 589-8142; TWX 7108250437 IEEECOMPSO.

Small Computer (R)evolution

compcon 84 fall

IEEE COMPUTER SOCIETY

Hyatt Regency Crystal City
Arlington, Virginia
September 16–20, 1984



Photo Courtesy of NASA

ADVANCE ANNOUNCEMENT

COMPCON FALL '84 will focus on the (R)evolutionary impact of rapidly evolving computer technology on the modern world. MICRO, MINI, and PERSONAL COMPUTERS are beginning to touch all aspects of our lives. They affect how we think, how we act, how we work, how we play, and even our health. Their influence is explicit as tools—for management, for science, for administration, and implicit as embedded controls in many other systems—automobiles, industrial robots, and consumer products.

Advancing computer hardware technology has driven down the cost, size, weight and power consumption to make small computers appropriate and affordable on a scale almost unthinkable as recently as a few years ago. New applications of small computers, from single chip micro processors to desk top minis, are expanding at an explosive rate. The scope of applications seems limited only by human imagination. The SMALL COMPUTER (R)EVOLUTION, COMPCON FALL '84 program will encompass a wide scope of applications: as tools for managers, professionals, non-professionals, students, home-users, hobbyists and as embedded elements of other systems. The program will include tutorials, panels, demonstrations, and technical papers.

Mini Tutorials will be held on Sunday September 16, 1984 from 1:00 p.m.–5:00 p.m.
- Introduction to Computer Applications
- End User Facilities
- Introduction to Computers
- Software Quality Assurance

Full Tutorials will be held on Monday September 17, 1984 from 9:00 a.m.–5:00 p.m.
- Local Network Technology
- Distributed Data Bases
- Ada
- C Language on Small Computers

Session topics will include (but not be limited to) new products, support software, application packages, user friendliness, embedded systems, hardware technology and architecture of small computer applications and end products. In addition to the world of applications (above), papers and demonstrations are invited on:

- Data/Computer Security
- Networking
- Electronic Mail
- Economic/Social Impact
- Graphics
- Robotics
 - CAD/CAM
 - Simulators
 - Computer Aided Instruction
 - Artificial Intelligence
 - Software Ownership Protection
 - Computers for the Handicapped

And other areas of interest in the realm of the SMALL COMPUTER (R)EVOLUTION

PRIZES WILL BE AWARDED FOR THE BEST PAPER ON NEW APPLICATIONS, THE BEST UNDERGRADUATE PAPER, AND THE BEST DEMONSTRATION OF THE USE OF SMALL COMPUTERS. Your papers and suggestions for special sessions, panels, tutorials and demonstrations are solicited.

SPECIAL EVENTS TAKING PLACE:

- **Vendor Exhibits**
- **Micro Computer Swap Meet**
 Sunday, September 16, 1984 from 10:00 a.m.–6:00 p.m. at Hyatt Regency Crystal City
- **Cocktail Party**
 Tuesday, September 18, 1984
- **Special Noontime Session**
 Wednesday, September 19, 1984
 Featured speaker and luncheon